Author:
Liu, Bin

Title:
Cryptographic Access Control

Security Models, Relations and Construction

Cryptographic Access Control: Security Models, Relations and Construction

By

Bin Liu

Department of Computer Science
University of Bristol

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering.

November 2019

Word count: 48337

Abstract

Traditional enforcement of access control policies heavily relies on reference monitors, which need to be run in trusted domains, be permanently online and mediate every access request from users. This inherent limitation directly impacts scalability and deployability of its applications. A solution to this problem is to employ cryptography, where policy enforcement depends on both security of the underlying cryptographic primitives and appropriate key distribution. This approach is known as cryptographic access control. It has the potential to reduce the reliance on monitors or even eliminate this need while enforcing the access control policies.

The existing works in cryptographic access control mainly focused on implementing various access control systems from basic cryptographic primitives and/or designing new primitives tailored for access control systems. However, the study on formal security models for cryptographic access control systems, which are of central importance, is usually neglected. Specifically, without formal security models, one cannot establish the link between security guarantees from cryptographic primitives and the enforcement of access control policies.

This problem was first addressed by Ferrara et al., whose recent work on cryptographic Role-Based Access Control (cRBAC) establishes rigorous foundations for the analysis of cryptographic access control systems. In this thesis, we continue their line of research. Our main contributions are definitional. We study security of cRBAC systems in both game-based and simulation-based settings, and the relations between the security notions. We also initiate the study of policy privacy in the context of cryptographic access control systems. The privacy issue does not arise in traditional monitor-based policy enforcement, but cryptographic access-control systems may inadvertently leak information on the underlying access control policies. Such information can be sensitive in many scenarios. Next, we propose a construction of cRBAC system which employs a new privacy-preserving encryption. Our security proofs confirm that our purposal securely enforces both read and write access to the file system, while preserving policy privacy to a certain degree. Finally, we study the efficiency implications of secure cRBAC systems. Our result shows that supporting permission revocation is inherently costly in such systems.

Acknowledgements

First and foremost, I would like to sincerely thank my supervisor Professor Bogdan Warinschi for his continuous support and guidance throughout my studies at University of Bristol. Thank you for guiding me on the path to become a researcher. Thank you for being patient with me and encouraging me when I am in difficulty.

I would like to express my special thanks to my wife. I really appreciate her pain and dedication over these years. She gave up the opportunity to study at a great university for a master's degree and decided to support me as a housewife. She gave birth to our little angel, Alice (a name that all cryptographers are familiar with). She is a great wife and mother. Without her unconditional support and help, I would not have finished my PhD study.

I would like to thank my mother for giving me everything. When my father passed away last year, I suddenly realised just how much I have always asked of her, and how much I owe her.

I am very grateful to those people who offered me their friendly help last year so that I can get home for my father's last moments. Thank you all. Many thanks to my friends Yan Yan, Si Gao and Zicheng Gui for their friendship and their help in times of need. I would also like to thank all the people in the crypto group.

Finally, may my father rest in peace.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE:.....

Contents

List of Figures

Chapter 1

Introduction

Traditional access control mechanisms heavily rely on reference monitors to enforce policies [3]. Since the reference monitors have to be executed in the trusted domains and be permanently online to mediate every access request from users, this approach has the inherent limitations that impact scalability and deployability of applications. Especially, it is not suitable for the emerging trend of outsourcing data storage to untrusted file storage servers where hosting a trusted monitor is almost impossible. An alternative solution is to employ cryptographic techniques to enforce access control policies, which is known as cryptographic access control. The idea behind is simple and elegant: the files are protected by cryptographic primitives, while the access control policies are enforced by appropriately providing the keys to the authorised users. It is a promising solution as cryptography is a natrual solution for preserving data confidentiality and integrity. More importantly, cryptographic enforcement of access control policies does not suffer from the limitations mentioned above. Therefore, cryptography can help to reduce the reliance on reference monitors and even to eliminate this need.

Previous results in cryptographic access control range from designing access control systems from basic cryptographic primitives [32, 1, 24, 22, 21, 18] to the more advanced cryptographic primitives tailored for access control [35, 50, 52, 30]. However, a primary concern of the most existing works is the absence of formal security models for the whole systems. Although cryptogrpahic primitives can protect data privacy at points, security of the primitives does not necessarily translate to security of the whole system. More precisely, the correct policy enforcement in cryptographic access control systems involves more subtle issues like appropriate key management/distribution and timely update of cryptographic materials. Without formal security models, one cannot establish the link between the implementation of the cryptographic access control system and the

specification of the policy being enforced. Thus many of the existing works do not offer any proof at all for their constructions [39, 59, 20, 56], meaning only informal security guarantees can be provided.

This problem was first addressed by Ferrara et al. in their recent work [28]. They showed how to use attribute-based encryption scheme to provably enforce Role-Based Access Control (RBAC) policy on read access to a file system. Particularly, they defined a precise syntax of the access control system and proposed a formal security model that captures secure read access to the file system within their framework. Their result comes with a construction that meets the proposed security notion, but write access to the file system is still handled by the reference monitor.

The work in this thesis continues the line of Ferrara et al.'s research and extends it in several directions. First, we further reduce the dependency on policy-enforcing monitors by supporting access control on write access. In our extended system model, users are allowed to have (quasi-)unrestricted write access to the files, but only those contents written by authorised users will be considered as valid. The monitor is therefore tasked with policy administration only. Based on this, we propose a formal security model with respect to secure write access for cRBAC systems.

We also address the policy privacy issues in the context of cryptographic access control. The correct enforcement of policies is the core requirement of cryptographic access control systems, yet policy privacy is not an ordinary security concern. In traditional access control, the policy being enforced is kept by the policy enforcer and only policy-compliant access request will be granted. Therefore, the information about the access control policy is perfectly hidden from users: they can only learn whether they have access to particular files or not. However, cryptographic implementations of access control systems may reveal more information than desirable. Any change to the policy being enforced will be directly reflected in the system global state, which means the publicly available information (e.g. metadata and the encrypted files) and even users' local states might unintentionally reveal information about the policy. Such informaiton can be critical in the areas where privacy is mandated by law or regulations (e.g. governments, enterprises, etc.) or it can be highly sensitive in some other areas (e.g. institutions, hospitals, etc.). In such settings, cryptographic access control may become unusable.

There have been many cryptographic primitives for preserving various forms of policies proposed [11, 31, 63, 9, 58], but these may not suffice to preserve policy privacy in the access control systems that employ them. Specifically, the absence of formal security

models could result in the impossibility of rigorously proving that such information is not revealed in the system. To this end, we propose different security notions to capture several distinct aspects of policy privacy. Our work can be considered as the first rigorous approach to policy privacy in cryptographic access control systems. Even though our results are in the RBAC model, they still can serve as an inspiration to the work in similar contexts.

As widely acknowledged, coming up with precise security models for complex systems turns out to be a tricky business. In order to appropriately model cryptographic enforcement of RBAC policies, so far we have already proposed several security models for different security properties in game-based setting. To step further towards the goal, we then turn to study cryptographic RBAC system in simulation-based setting, where security is defined by requiring the information revealed during the execution of a system is at most as much information revealed by an ideal version of the system. This type of security notions is intuitive but often cumbersome to work with. Since the idealised system preserves all security properties expected of a given cryptographic task, the real system which is considered to be secure under this paradigm therefore inherits all those security properties. For cryptographic RBAC systems, the idealised version is exactly the correct enforcement of RBAC policies. Therefore, there is no need to enumerate all security properties separately and to worry about if a system that holds all those security properties can cryptographically enforce the RBAC policy as expected. Moreover, simulation-based security with composability property is highly desirable in cryptographic access control due to its applicability. Cryptographic access control systems need to maintain their security guarantees when employed within different higher level protocols.

We propose the first simulation-based security notion for cryptographic RBAC systems within the Universal Composable (UC) security framework [12]. Then we study its relation with the existing game-based security notions. The result shows our new security notion is strictly stronger than the existing ones with respect to secure access. We also identify a gap between the simulation-based security and the game-based security. More precisely, we show that there exists no UC-secure cRBAC system with adaptive corruptions, even given access to secure channels and an idealised versioning file system.

Inspired by the study of the relation between the two types of security models for cRBAC systems, we identify two different attacks which are not captured by the existing game-based security notions. Therefore, we propose two new security notions of secure

read and write access respectively. The new security notion for read security is called *past confidentiality* which is strictly stronger than the existing one. Interestingly, we found that the recent results on cryptographically access control systems fall short to this security property, even though their constructions were proven to securely enforce the access control policies within their individual frameworks. The other one for write access is called *local correctness* and serves as a complementary notion to the existing notion of secure read access.

We then propose a construction of cRBAC system that enforces both read and write access to a file system. The main ingredient of our construction is a variant of Predicate Encryption (PE) scheme called Predicate Encryption with Specific Public Keys (PE-SK). It allows our construction to preserve a certain degree of privacy for the policy being enforced. Our proofs confirm that the construction securely enforces access control on both read and write access to a file system, while preserving a certain degree of policy privacy.

Finally, we present some theoretic results with respect to the lower bounds for secure cRBAC systems. By lower bound for secure cRBAC systems, we mean the intrinsic computation overheads of cRBAC systems which securely enforce RBAC policy with respect to read and write access.

1.1 Related Work

The enforcement of access control policies with the use of cryptographic techniques has received considerable attention in recent decades. Gudes' work in 1980 [37] can be seen as the seminal work in cryptographic access control. He showed how to use cryptography to enforce different protection policies on a local file system and also suggested some basic design principles of the use of cryptographic schemes. However, his result does not include a concrete construction of the access control system and he does not consider the key management problem in such systems. Later, the works of Gifford's [32] and Akl et al.'s [1] addressed the key-management problem in cryptographic access control but policy update was not considered. Harrington and Jensen discussed the infeasibility of employing traditional monitor-based access control on distributed file systems and suggests to use cryptographic techniques to enforce the access control policies [39]. However, the access control system they proposed only uses cryptography to implement partial access control mechanism rather than to enforce the access control policies.

Recently, with the development of advanced cryptographic primitives such as Identity-

Based Encryption (IBE) [8], Predicate Encryption (PE) [47] and Attribute-Based Encryption (ABE) [36, 7] which are well-suited for enforcing different access control policies, there have been significant works on cryptographic access control. Crampton has shown that cryptography can be used to enforce RBAC policy by re-writing RBAC policies as information flow policies and applying the key assignment scheme accordingly. He also examined the connection between his cryptographic role-based access control scheme and ABE [22]. Later, Campton showed that general interval-based access control policies can be enforced using key assignment schemes [23]. Zhu et al. proposed an access control system based on role-key hierarchy model and designed new signature and encryption schemes (both are pairing-based) which are tailored for the system they proposed [70]. There also have been many other similar works on cryptographic access control systems or on designing new primitives for them [17, 18, 16, 41, 19, 66]. The common problem of these works is the absence of formal security definitions for the whole system, which will lead to a worrying situation where only informal security proofs can be provided.

Halevi et al. proposed the first formal security definition for access control in distributed file storage system [38]. However, their security definition is for a specific protocol rather than for a general one. Ferrara et al. formally defined security for cryptographic Role-Based Access Control (cRBAC) systems with respect to read access [28]. They also provided a construction of the access control system based on a variant of Predicate Encryption scheme, and showed that the security notion can be provably achieved. Later, Alderman et al. followed this line of research. They proposed formal security definitions for cryptographic enforcement of information flow policies (on read access only) and provided a construction which is proven to be secure with respect to their security definitions [2]. However, their security definition for read security does not capture the security concern of retrieving the previous file contents in an unauthorised manner (which will be discussed in Session 4.3.2).

Garrison III et al. studied the practical implications of cryptographic access control systems that enforces RBAC policies [43]. They analysed the computational costs of two different constructions of cryptographic role-based access control systems via simulations with the use of real-world datasets. Their result shows that supporting for dynamic access control policy enforcement may be prohibitively expensive, even under the assumption that write access is enforced with the minimum use of reference monitors.

1.2 Contributions

In this thesis, we mainly focus on formal security models of cryptogrpahic Role-Based Access Control (cRBAC). We highlight our main contributions as follows.

1. We propose several formal security models for cRBAC systems to precisely model cryptographic enforcement of RBAC policy.

2. We address the policy privacy issues in the context of cryptographic access control and propose formal security models for different flavours of policy privacy. Our work can be considered as the first rigorous approach to policy privacy in cryptographic access control systems.

3. We propose a construction of cRBAC system based on a variant of predicate encryption, and formally prove that our construction meets the existing security notions.

4. We study security of cRBAC systems in UC framework. We propose a security notion for cRBAC systems in UC framework and show that this notion is strictly stronger than the existing ones. We also identify a gap between simulation-based and game-based security: it is impossible for a cRBAC system to be UC-secure with adaptive corruptions.

5. We show some lower-bounds for secure cRBAC systems.

1.3 Outline of the Thesis

The thesis is organised as follows:

The preliminaries are presented in Chapter 2. We include the notations and the relevant background which are required to understand the remainder of this thesis.

In Chapter 3, we introduce our notion of a cRBAC system, which extends the notion introduced by Ferrara et al. in [28] by allowing authorised users to perform write operations on files.

In Chapter 4, we tease out the security properties expected from a cRBAC system that correctly enforces the policy and formalise them in the game-based setting. Specifically, we redefine the two existing security definitions: correctness and secure read access in our current system model and introduce a security definition with respect to write access. After having studied security of cRBAC systems in simulation-based setting, we

identify two different types of attacks which are not captured by the existing security definitions. Therefore, we propose two new security definitions: past confidentiality and local correctness. The former provides stronger security guarantee on read access, while the latter serves as complementary to the definition of secure write access. We also start to address the issue of policy privacy, which is another important feature in cryptographic access control systems. We identify and formalise several different flavours of policy privacy targeting to systems with different privacy demands. Finally, we present a construction of cRBAC system which is built on a PE-SK scheme. Our construction securely enforces both read and write access to a file system, while preserving a certain degree of policy privacy.

In Chapter 5, we study security of cRBAC systems in simulation-based setting. Our first result is a formal security definition for secure cRBAC systems in the UC framework. Then we study the relation between our UC security definition and the existing security definitions for cRBAC systems. Finally, we show an impossibility result of the UC secure cRBAC system with adaptive corruptions.

In Chapter 6, we show some lower bounds for secure cRBAC systems. We mathematically show that the support of dynamic policy update can be costy in secure cRBAC systems.

In Chapter 7, we make conclusions from our results and discuss the possible directions for future work.

1.4 Publications

Here we list all the publications related to the work presented in the thesis.

[27] Anna Lisa Ferrara, Georg Fuchsbauer, Bin Liu, and Bogdan Warinschi. Policy privacy in cryptographic access control. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 4660, 2015.

[51] Bin Liu and Bogdan Warinschi. Universally composable cryptographic role-based access control. In *Provable Security - 10th International Conference, ProvSec 2016, Nanjing, China, November 10-11, 2016, Proceedings*, pages 6180, 2016.

Chapter 2

Preliminaries

In this chapter we introduce all of the notations and the foundational concepts which will be used in the rest of this thesis.

2.1 Notations

For assignment, we use $x \leftarrow y$ to denote that we assign x the value of y. If S is a set, $x \leftarrow S$ denotes that we assign x the value of a member in S, where the member is chosen uniformly at random. If A is an algorithm, $x \leftarrow A(y)$ denotes the assignment of x with the output of running A on the input y. If A is a randomised algorithm, then we use $x \leftarrow\!\!\$\, A(y)$ to denote the assignment.

If s is a string, $|s|$ denotes its length. If S is a set, $|S|$ denotes its size. ϵ denotes the empty string. $\perp$ denotes an error, its meanning depends on the content: it could be decryption failure or an error returned by the oracle. If k is an integer, 1^k denotes the string of k 1s.

Negligible function. In cryptography, security are usually defined by requiring some "bad event" to happen with very small probability. When using a function to represent a probability, we say the function is negligible if it tends to zero faster than the inverse of any polynomial.

Definition 1 (Negligible Function)**.** *A function $\mu : \mathbb{N} \to \mathbb{R}$ is **negligible** if for every positive polynomial p there exists an integer N such that for every integer $n > N$, $\mu < \frac{1}{p(n)}$.*

We say a function μ is **non-negligible**, if there exists a polynomial p, there exists an integer N such that there exists an integer $n > N$, $\mu \leq \frac{1}{p(n)}$.

2.2 Provable Security

In 1984, a paradigm was proposed by Goldwasser and Micali in their seminal paper probabilistic encryption [34] which was later known as provable security. This approach relates the security of a scheme to some specified mathematically intractable problem. The proof of security can be given by reduction. Any efficient algorithm that breaks the security of the scheme can be used to solve the mathematical problem. However, if the problem is really intractable, we get a contradiction. Therefore such an efficient algorithm does not exist and we prove the security of the scheme.

2.3 Security Definitions

In order to reason about the security of a protocol, a security definition must be available. As a central task of provable security, establishing appropriate security definitions is of great importance. Typically, there are two main definitional approaches to capture security requirements of protocols.

2.3.1 Game-Based Security

The so-called game is a conceptualisation of the interactions between the protocol (or scheme) and an adversary who attacks the protocol. The game specifies some goal for the adversary to achieve, which is usually posed by a hypothetical challenger. The goal precisely clarifies what constitutes an attack against the protocol. The adversary may further get access to some oracles, which will provide it some information that it can obtain when attacking the protocol. Any oracle call that will lead to a trivially win is always prohibited. Security defined via this approach demands that no efficient adversary can achieve its goal with probability exceeding some threshold particularly the probability of winning by chance.

The security of numerous cryptographic schemes and protocols have been defined by this approach (e.g. public key encryption[34], key exchange [5, 15], etc.). The most appealing advantage of this approach is its relative simplicity: executions only consider stand-alone scenarios where the protocol is in complete isolation from others, and different security goals (e.g. privacy and integrity of sensitive data) are treated independently from one another. However, the main concern is the information that an adversary can learn when attacking the protocol must be specified in the game. The threat from the environment that the protocol is being employed might not be

appropriately captured. One solution is to explicitly define the security for specific environments. But in such case the proofs can hardly provide security guarantee when the protocol is employed in the environments that have not been considered. In addition, it may not always be possible to exhaustively enumerate the different properties that one may desire from a system of a certain degree of scale.

2.3.2 Simulation-based Security

An alternative approach to define security is based on the simlulation paradigm. The root of this approach goes back to Goldreich, Micali and Wigderson's paper [33]. It is also known as the *real/ideal-world* paradigm. Security is defined by comparing a system with an idealized version and demands that the real execution of a system reveals at most as much information is revealed by an ideal version of the system. Canetti's UC-framework [12], Pfitzmann and Waidner's composed system [55] are of this paradigm.

Simulation-based security definitions offen demand stringent requirements which might lead to inefficient implementations of some cryptographic tasks, or even prevent the implementations - even if some protocols seem to be secure for practice purposes, they might not be secure under simulation-based definitions (e.g. UC-secure commitment scheme [13]). In addition, some tasks cannot be proved to be secure in simulation-based settings (e.g. the non-commitment encryption with adaptive corruptions in the plain model [54]).

2.4 The Universal Composability(UC) Framework

The real/ideal-world paradigm has been further developed by the UC framework. In the UC framework, the trusted party of the ideal process is modelled as an entity called *ideal functionality* and denoted by $\mathcal{F}$. In addition to handling the inputs obtained repeatedly from the parties and generating the prescribed outputs, $\mathcal{F}$ is allowed to interact with the adversary, in a way that captures the allowed adversarial influence and information leakage of the protocol. To provide security guarantee under composition, the UC framework introduces an adversarial entity called the *environment* $\mathcal{Z}$, which represents all possible settings in which the protocol can be executed. $\mathcal{Z}$ acts as an interactive distinguisher which aims to tell if it is interacting with the real protocol or with the ideal one. In the process, the environment is allowed to exchange information with the adversary, to provide inputs to the participants of it choice and to obtain outputs from them. A protocol Π is said to securely realise the functionality $\mathcal{F}$, if for any adversary

$\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that no environment can distinguish between its interactions with parties running Π and $\mathcal{A}$ and the interactions with the ideal process for $\mathcal{F}$ and $\mathcal{S}$.

An special type of adversary is the so-called *dummy adversary* $\mathcal{D}$. This adversary simply delivers the messages from the environment to the parties and forwards the messages from the parties to the environment: this adversary essentially allows the environment to fully control the input/output and the communication between the parties. A simulator that works for the dummy adversary essentially gives rise to a simulator for any other adversary.

An important concept in the UC framework is the *hybrid model*, an execution setting which is a mix between a real protocol and an idealised setting. Specifically, in an $\mathcal{F}$-hybrid the parties running the protocol can use multiple copies of an ideal functionality $\mathcal{F}$. The extension of the notion of realizing of an ideal functionality in the hybrid model is immediate. In fact, it captures the essence of the general composition theorem specific to UC. If a protocol ρ securely realises an ideal functionality $\mathcal{G}$ in $\mathcal{F}$-hybrid model and there is a protocol π securely realises $\mathcal{F}$, then the composed protocol $\rho^{\pi/\mathcal{F}}$ where all the calls to $\mathcal{F}$ are replaced by calls to π securely realises $\mathcal{G}$. Hence π provides the same security guarantee as the ideal functionality $\mathcal{F}$ even if used within an arbitrary protocol ρ; furthermore the composed protocol $\rho^{\pi/\mathcal{F}}$ still provides the same security guarantee as the ideal functionality $\mathcal{G}$.

One particular application of hybrid models is to capture various communication models. This is achieved by formulating an appropriate ideal functionality $\mathcal{F}$ that represents the abstraction from the communication, then real-world protocols in the communication model can be presented in the $\mathcal{F}$-hybrid model. To exemplify this approach, we present $\mathcal{F}_{\text{SMT}}$, the ideal functionality for secure message transmission (aka secure communication) in Fig. 2.1. In $\mathcal{F}_{\text{SMT}}$, a sender P_S with input m sends its input to a receiver P_R, while the adversary only learns $|m|$, the length of the message m, and can delay the message delivery. Notice that $\mathcal{F}_{\text{SMT}}$ can only transmit a single message, to transmit multiple messages we need to use multiple instances of $\mathcal{F}_{\text{SMT}}$. We refer to [12] for more details and formal descriptions about the UC framework.

2.5 Digital Signature

The concept of digital signature schemes was first introduced by Diffie and Hellman in [25]. A digital signature aims to provide trust on message integrity, authentication and

Functionality $\mathcal{F}_{\text{SMT}}$

$\mathcal{F}_{\text{SMT}}$ proceeds as follows, with a sender P_S, a receiver P_R and an adversary $\mathcal{S}$.

1. Upon receiving an input $(\texttt{Send}, sid, P_R, m)$ from P_S, send $(\texttt{Sent}, sid, P_S, P_R, |m|)$ to the adversary and generate a delayed output $(\texttt{Sent}, P_S, sid, m)$ to P_R then halt.
2. Upon receiving $(\texttt{Corrupt}, sid, P)$ from $\mathcal{S}$, where $P \in \{P_S, P_R\}$, reveal m to the adversary. If $P = P_S$ and the message has not yet been sent to P_R, then ask $\mathcal{S}$ for a value m' and output $(\texttt{Sent}, P_S, sid, m')$ to P_R then halt.

Figure 2.1: Ideal functionality for secure message transmission, $\mathcal{F}_{\text{SMT}}$.

also non-repudiation. The following definition of digital signature schemes is adapted from [46].

Definition 2. *A digital signature scheme Σ is given by a tuple of probabilistic polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *that satisfy the following:*

1. *The randomised key-generation algorithm* KeyGen *takes as input a security parameter 1^λ and outputs a pair of keys (vk, sk), where vk is the verification key and sk is a secret signing key.*
2. *The randomised signing algorithm* Sign *takes as input a secrect signing key sk and a message m and outputs a signature σ.*
3. *The deterministic verification algorithm* Verify *takes as input a verification key vk, a message m and a signature σ and outputs a bit b, where $b = 1$ if σ is a valid signature of m under vk and $b = 0$ otherwise.*

The Existentially Unforgeable under Chosen Message Attacks (EU-CMA) security is considered as the standard security requirement for digital signature schemes. It is defined through the experiment $\mathbf{Exp}^{\text{eu-cma}}_{\Sigma,\mathcal{A}}$ which involves an a digital signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and adversary $\mathcal{A}$. In the experiment, a pair of keys (vk, sk) is generated by running the key generation algorithm. Then $\mathcal{A}$ is handed the verification key vk and has access to a signing oracle Sign. Eventually, the adversary terminates with an output (m^*, σ^*), where m^* is a message of its choosing and σ^* is the signature. It wins the game if m^* has never been queried to Sign and σ^* is a valid signature of m^* under vk.

Definition 3 (EU-CMA)**.** *A digital signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is Existentially Unforgeable under Chosen Message Attacks if for any probabilistic polynomial-*

time adversary $\mathcal{A}$, it holds that

$$\mathbf{Adv}^{eu\text{-}cma}_{\Sigma,\mathcal{A}}(\lambda) := \Pr\left[\mathbf{Exp}^{eu\text{-}cma}_{\Sigma,\mathcal{A}}(\lambda) \to \texttt{true}\right]$$

is negligible in λ, where $\mathbf{Exp}^{eu\text{-}cma}_{\Sigma,\mathcal{A}}$ is defined as follows:

$\underline{\mathbf{Exp}^{eu\text{-}cma}_{\Sigma,\mathcal{A}}(\lambda)}$

$L \leftarrow \emptyset$

$(vk, sk) \leftarrow\!\!\$\ \mathsf{KeyGen}(\lambda)$

$(m^*, \sigma^*) \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, vk : \textsc{Sign})$

if $m^* \notin L \wedge \mathsf{Verify}(vk, m^*, \sigma^*) = 1$ **then**

  return `true`

else return `false`

$\underline{\textsc{Sign}(m)}$

$L \leftarrow L \cup \{m\}$

return $\sigma \leftarrow\!\!\$\ \mathsf{Sign}(sk, m)$

2.6 Predicate Encryption with Specific Public Keys

In classical predicate encryption [47], ciphertexts are encrypted with identities (sets of attributes) and secret keys correspond to predicates. A ciphertext associated with an identity I can be decrypted by a secret key corresponding to a predicate f only when $f(I) = 1$ is satisfied, whereas the identity I must be given explicitly as the input of the encryption algorithm. This very nature of predicate encryption reveals the identity associated to the ciphertext explicitly. Thus, we need a way to hide the identity during encryption.

In [27], we overcome this by introducing a variant of predicate encryption called predicate encryption with (identity-) specific public keys (PE-SK). It allows for generating public key with any identity, which can be used to encrypt a message instead of providing the identity explicitly in the encryption algorithm, while the obtained ciphertext can be decrypted by the users with secrect keys for predicates which hold on the identity.

Definition 4. *A Predicate Encryption with Specific Public Keys (PE-SK) scheme $\mathcal{PE}$ is given by a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Setup}, \mathsf{PKGen}, \mathsf{DKGen}, \mathsf{Enc}, \mathsf{Dec})$*:*

1. *The randomised setup algorithm* Setup *on input the security parameter λ (and*

optional parameters such as the attribute universe) returns a pair (mpk, mdk) *of a master public and master secret (decryption) key.*

2. *The randomised public-key generation algorithm* PKGen *on inputs* mpk *and* I *returns a public encryption key* pk_I *for identity* I*.*

3. *The randomised decryption-key generation algorithm* DKGen *on inputs* mdk *and a predicate* f *returns a decryption key* dk_f *for* f*.*

4. *The randomised encryption algorithm* Enc *on inputs* pk_I *and* m *returns a ciphertext* c*.*

5. *The deterministic decryption algorithm* Dec *on inputs* sk_f *and a ciphertext* c *returns a string* m *(or* $\perp$*).*

Correctness A PE-SK scheme $\mathcal{PE}$ is *correct* if for all λ, f, I, m, r, all (mpk, mdk) output by $\mathsf{Setup}(1^\lambda)$, all pk_I output by $\mathsf{PKGen}(mpk, I)$ and all dk_f output by $\mathsf{SKGen}(mdk, f)$ we have $\mathsf{Dec}(dk_f, \mathsf{Enc}(pk_I, m; r)) = m$ if and only if $f(I) = 1$. Since when knowing mdk one can always derive a key and then decrypt, we also directly write $\mathsf{Dec}(mdk, c)$.

Identity-hiding public keys We first introduce a security notion that formalises the requirement that keys do not reveal for which identity they are. An adversary must guess a random bit b after getting the master public key mpk and access to a challenge oracle LR, which on input (I_0, I_1) returns an encryption key for I_b. Note that this also formalises the fact that an adversary cannot tell whether two keys are for the same identity: given mpk, it can produce a key for pk_{I_0} and being given pk_{I_b} guess b by linking keys.

The adversary is also provided a DKGen oracle, which models collusions between users. To prevent trivial attacks, we require the following restriction. When queried on f, DKGen first checks whether $f(I_0) = f(I_1)$ for all (I_0, I_1) queried to LR (otherwise the decryption key could be used to test whether a ciphertext produced with the challenge key pk_{I_b} decrypts correctly or not). Analogously, LR only answers queries (I_0, I_1) if $f(I_0) = f(I_1)$ for all f queried to DKGen (otherwise the adversary can use the decryption key to test whether a ciphertext produced with the challenge I_b decrypts correctly or not).

Definition 5 (Identity-hiding public keys)**.** *The following game formalises the security requirement that ID-specific public keys do not reveal any non-trivial information about the identities they are for:*

$\underline{\mathbf{Exp}^{id\text{-}h\text{-}pk}_{\mathcal{PE},\mathcal{A}}(\lambda)}$

$b \leftarrow\!\!\$\ \{0,1\}; F \leftarrow \emptyset; Ch \leftarrow \emptyset$

$(mpk, mdk) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$

$b' \leftarrow\!\!\$\ \mathcal{A}(mpk : \text{DKGen}, \text{LR})$

Return $(b' = b)$

$\underline{\text{DKGen}(f)}$

For all $(I_0, I_1) \in Ch$*:*

If $f(I_0) \neq f(I_1)$ *then Return* $\perp$

$F \leftarrow F \cup \{f\}$

Return $dk_f \leftarrow\!\!\$\ \mathsf{DKGen}(mdk, f)$

$\underline{\text{LR}(I_0, I_1)}$

For all $f \in F$*:*

If $f(I_0) \neq f(I_1)$ *then Return* $\perp$

$Ch \leftarrow Ch \cup \{(I_0, I_1)\}$

Return $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, I_b)$

We say a PE-SK scheme $\mathcal{PE}$ *has* identity-hiding encryption keys *if for any p.p.t. adversary* $\mathcal{A}$*,*

$$\mathbf{Adv}^{id\text{-}h\text{-}pk}_{\mathcal{PE},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{id\text{-}h\text{-}pk}_{\mathcal{PE},\mathcal{A}}(\lambda) \to 1] - \tfrac{1}{2} \right|$$

is negligible in λ*.*

Message-hiding While our first security notion ensures that public keys (and ciphertexts created from them) do not reveal their associated identity, the second notion formalises the traditional requirement that ciphertexts of different messages should be indistinguishable. In contrast to the first notion, this also exists for standard PE, where this is termed as *payload-hiding* [47].

This notion is formalised via a game where the adversary must distinguish messages encrypted under a key whose corresponding secret key it must not know. We give the adversary access to an oracle that generates public keys pk_I for I of the adversary's choice. The adversary then chooses one such key and two equal-length messages (M_0, M_1) and

gets an encryption of M_b under that key.

More formally, the game stores queried keys pk_I and the identity I in the first empty index of two lists PK and $\mathcal{I}$, respectively. When the adversary asks for a challenge under the j-th key by querying (j, M_0, M_1), it receives an encryption of M_b under $PK[j]$. The corresponding identity $\mathcal{I}[j]$ is then added to the list of challenges Ch.

The adversary can also query decryption keys for any predicate f, which is then added to a list F. To prevent trivial attacks, the experiment maintains the invariant that for all $f \in F$ and $I \in Ch$ it should hold that $f(I) = 0$; otherwise, if for some f and I we had $f(I) = 1$, the adversary could query a challenge under the key for I and then decrypt it using dk_f.

Definition 6 (Message hiding)**.** *The following game formalises the fact that ciphertexts hide the encrypted message:*

$\underline{\mathbf{Exp}^{msg\text{-}hide}_{\mathcal{PE},\mathcal{A}}(\lambda)}$

$b \leftarrow\!\!\$\ \{0,1\}$; $ctr \leftarrow 1$; $PK, \mathcal{I}, F, Ch \leftarrow \emptyset$

$(mpk, mdk) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$

$b' \leftarrow\!\!\$\ \mathcal{A}(mpk : \text{PKGen}, \text{DKGen}, \text{LR})$

Return $(b' = b)$

$\underline{\text{PKGen}(I)}$

$pk_I \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, I)$

$\mathcal{I}[ctr] \leftarrow I$; $PK[ctr] \leftarrow pk_I$; $ctr \leftarrow ctr + 1$

Return pk_I

$\underline{\text{DKGen}(f)}$

For all $I \in Ch$*:*

If $f(I) = 1$ *then Return* $\perp$

$F \leftarrow F \cup \{f\}$

Return $dk_f \leftarrow\!\!\$\ \mathsf{DKGen}(mdk, f)$

$\underline{\text{LR}(j, M_0, M_1)}$

If $|M_0| \neq |M_1|$ *then Return* $\perp$

Let $(pk_I, I) \leftarrow (PK[j], \mathcal{I}[j])$

For all $f \in F$*:*

If $f(I) = 1$ *then Return* $\perp$

$Ch \leftarrow Ch \cup \{I\}$

Return $C \leftarrow\!\!\$\ \mathsf{Enc}(pk_I, M_b)$

We say a PE-SK scheme $\mathcal{PE}$ *has* message-hiding ciphertexts *if for any probabilistic polynomial-time adversary* $\mathcal{A}$*,*

$$\mathbf{Adv}^{msg\text{-}hide}_{\mathcal{PE},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{msg\text{-}hide}_{\mathcal{PE},\mathcal{A}}(\lambda) \to 1] - \tfrac{1}{2} \right|$$

is negligible in λ*.*

2.7 Role-Based Access Control

Role-Based Access Control (RBAC) is a general access control model that has been widely adopted in various systems. It simplifies the management on users' permissions by introducing an indirection, namely the roles [61, 60, 26]. Roles are the central concept of RBAC, since the policies are constructed around roles. The RBAC policies are decomposed into two assignments: the user-role assignment and the permission-role assignment. Both of the assignments can be managed seperately. A user is authorised to a permission if there exists a role of the users' has been assigned with the permission. In this thesis, we will only focus on the core RBAC [60].

An RBAC system consists of:

- U: a finite set of users
- R: a finite set of roles
- O: a finite set of objects
- P: a finite set of permissions where each permission is an *object-operation* pair
- $UA \subseteq U \times R$: a relation modelling the user-role assignment
- $PA \subseteq P \times R$: a relation modelling the permission-role assignment

We denote the *read* permission and the *write* permission of a file $o \in O$ by $(o, \texttt{read})$ and $(o, \texttt{write})$ respectively. Follow the work of [28], we assume that the set of roles R is fixed due to the fact that the role structures in organisations are usually stable. Therefore, at any point the state of a RBAC system over a fixed role set R is a tuple $S = (U, O, P, UA, PA)$. We summarise the typical administrative RBAC commands and their descriptions in Figure 2.2.

Command	Description
$\mathsf{AddUser}(u)$	Add a new user u to the system
$\mathsf{DelUser}(u)$	Remove an existing user u from the system
$\mathsf{AddObject}(o)$	Add a new object o to the system
$\mathsf{DelObject}(o)$	Remove an existing object o from the system
$\mathsf{AssignUser}(u, r)$	Assign the user u to the role r
$\mathsf{DeassignUser}(u, r)$	Deassign the user u from the role r
$\mathsf{GrantPerm}(p, r)$	Grant the permission p to the role r
$\mathsf{RevokePerm}(p, r)$	Revoke the permission p from the role r

Figure 2.2: Administrative RBAC commands.

We describe an RBAC system in terms of a state-transition system. Let RULES be the set of state-transition rules correspond to the administrative RBAC commands which are specified in Figure 2.2, given two states $S = (U, O, P, UA, PA)$ and $S' = (U', O', P', PA', UA')$, there is a *transition* from S to S' with command $q \in$ RULES denoted $S \xrightarrow{q}_{\mathcal{S}} S'$ if one of the following conditions holds:

- [AddUser(u)]: $q = (\mathsf{AddUser}, u)$, $u \notin U$, $U' = U \cup \{u\}$, $O' = O$, $P' = P$, $PA' = PA$ and $UA' = UA$;
- [DelUser(u)]: $q = (\mathsf{DelUser}, u)$, $u \in U$, $U' = U \setminus \{u\}$, $O' = O$, $P' = P$, $PA' = PA$ and $UA' = UA \setminus \{(u, r) \in UA \,\|\, r \in R)\}$;
- [AddObject(o)]: $q = (\mathsf{AddObject}, o)$, $o \notin O$, $O' = O \cup \{o\}$, $U' = U$, $P' = P \cup \{(o, \texttt{read}), (o, \texttt{write})\}$, $PA' = PA$ and $UA' = UA$;
- [DelOject(o)]: $q = (\mathsf{DelObject}, o)$, $o \in O$, $O' = O \setminus \{o\}$, $U' = U$, $P' = P \setminus \{(o, \cdot)\}$, $PA' = PA \setminus \{((o, \cdot), r) \in PA \,\|\, r \in R)\}$ and $UA' = UA$;
- [AssignUser(u,r)]: $q = (\mathsf{AssignUser}, (u, r))$, $u \in U$, $r \in R$, $U' = U$, $O' = O$, $P' = P$, $PA' = PA$ and $UA' = UA \cup \{(u, r)\}$;
- [DeassignUser(u,r)]: $q = (\mathsf{DeassignUser}, (u, r))$, $u \in U$, $r \in R$, $U' = U$, $O' = O$, $P' = P$, $PA' = PA$ and $UA' = UA \setminus \{(u, r)\}$;
- [GrantPerm(p,r)]: $q = (\mathsf{GrantPerm}, (p, r))$, $p \in P$, $r \in R$, $U' = U$, $O' = O$, $P' = P$, $PA' = PA \cup \{(p, r)\}$ and $UA' = UA$;
- [RevokePerm(p,r)]: $q = (\mathsf{RevokePerm}, (p, r))$, $p \in P$, $r \in R$, $U' = U$, $O' = O$, $P' = P$, $PA' = PA \setminus \{(p, r)\}$ and $UA' = UA$.

A run of an RBAC system is any finite sequence of transitions $S_0 \xrightarrow{q_0}_{\mathcal{S}} S_1 \xrightarrow{q_1}_{\mathcal{S}}$

$\ldots \xrightarrow{q_n}_{\mathcal{S}} S_{n+1}$, where S_0 is an *initial* state of the RBAC system.

A predicate $\mathsf{IsValid}(Cmd, arg)$ reflects that the execution of an RBAC administrative command $q = (Cmd, arg)$ is valid for the current state S. It is defined as follows:

$$\mathsf{IsValid}(Cmd, arg) \Leftrightarrow q \in \texttt{RULES} \wedge \exists S' : S \xrightarrow{q}_{\mathcal{S}} S'.$$

A predicate $\mathsf{HasAccess}(u, p)$ reflects that a user u has symbolically access to a permission p. It is defined as follows:

$$\mathsf{HasAccess}(u, p) \Leftrightarrow \exists r \in R : (u, r) \in UA \wedge (p, r) \in PA.$$

Chapter 3

Cryptographic Role-Based Access Control

The content presented in this chapter is adapted from the paper *Policy Privacy in Cryptographic Access Control* [27]. My contribuitons to the paper will be introduced in the next chapter.

3.1 Introduction

The notion of a cryptographic RBAC system (cRBAC) was first introduced in [28] where read access to a file system is controlled using cryptography, while write access is monitored on-line by the manager. We extend their notion by allowing authorised users to execute write operations on files, thereby foregoing completely the need for on-line monitors.

Concretely, we extend the cRBAC system in [28] to a setting where the users have (quasi) unrestricted read/write access to the file system and the manager is now tasked with the administration of access control policies only. Consider that if users are provided unrestricted write access to the file system, no amount of cryptography can prevent a malicious user to simply overwrite the existing contents. To this end, we propose using versioning file storage where users can only append contents but not delete any. These appends are then interpreted as logical writes to files. In practice, such a file system can be implemented with the use of log-structured techniques [64, 65]. In fact, our system model can be considered as a general model for cryptographic RBAC systems due to the features of the versioning file storage: the cryptographically protected files are always publicly accessible to all the users in the system and the file system does not implement

any access control mechanism, namely the enforcement of access control policy solely relies on cryptography.

In the published paper [27], the extended cryptographic RBAC system (cRBAC) is denoted by w-cRABC in order to be distinguished from the previous notion which only enforces read access. In this thesis, by a slight abuse of notation, we still denote it by cRBAC.

To summarise, in this chapter we make the following contributions:

- We present the notion of the extended cRBAC system that enforces access control on both read and write access to the file system.
- We provide a formal definition of cRBAC schemes.

3.2 System Model

The system model of a cRBAC is depicted in Figure 3.1. It involves three main entities: a *manager*, a *file system* and a set of *users*. The *manager* is assumed to be a trusted party and it is tasked with the administration of access control policies. Specifically, it is in charge of excuting RBAC administrative commands outlined in Session 2.7. The implementation of those commands involves key management and data encryption/re-encryption. In traditional monitor-based access control systems (depicted in Figure 3.2), the policy enforcer (the reference monitor) has to mediate every access request such that only the authorised requests (according to the access control policy being enforced) will be granted. In fact, the reference monitor needs to be involved in both policy administration and also the access to the files. Here, the manager of a cRBAC system is only responsible for the policy administration and does not involve in any access operation.

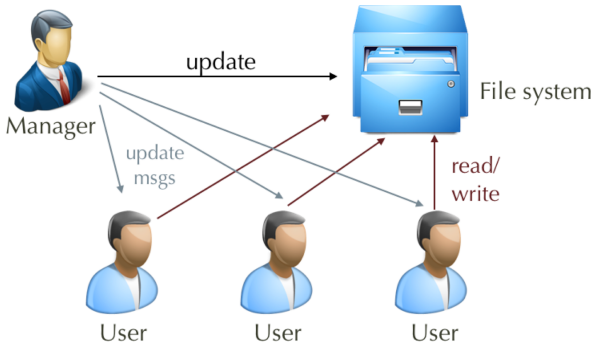

Figure 3.1: The system model of cRBAC.

The *file system* of a cRBAC is an untrusted storage that stores the files being enforced access control on. Unlike the file system of a traditional access control system which is in the protection domain and is controlled by the inherently centralised reference monitor, the file system of a cRBAC is assumed to be publicly accessible to the users (such as cloud storage). In implementation, it contains arrays of encrypted files and the related metadata. The file system itself does not implement any access control mechanism, but it must guarantee the availablity of data it stores. To support writing to the files by users, we require that the file system provide some extra guarantee to prevent malicious users from simply overwriting files and causing denial-of-service. We purpose the use of versioning storage, where the users can only write to the files by appending new versions to them instead of overwriting the existing contents. As the data owner, the manager could have richer interfaces to the file system than the users have and is therefore able to overwrite the file contents and to add/delete files.

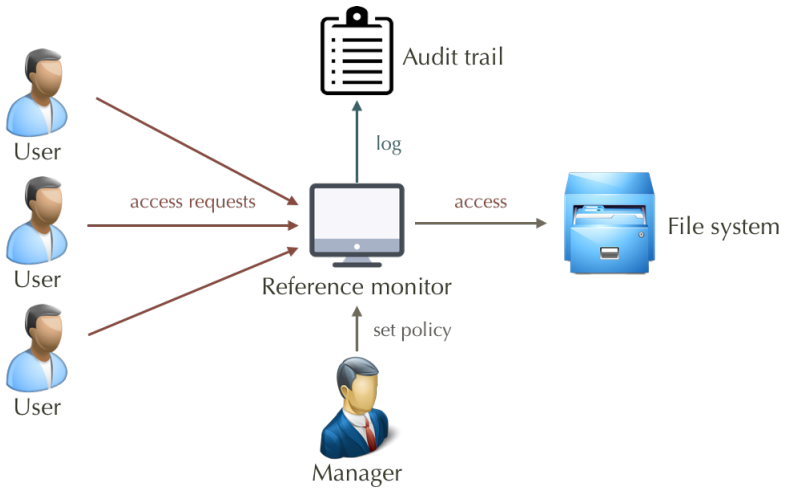

Figure 3.2: The system model of traditional monitor-based access control.

The *users* can get read and write accesses to the file system directly. When reading a file, one should first verify and fetch the most recent "valid" version (the file system should also guarantees correct ordering of file versions and the validity of a file version is guaranteed by cryptographic primitives) and then retrieve that as the current content by carrying out some decryption (if it holds the appripriate keys).

Secure channels are assumed between each of any two entities (but not between the users). For simplicity, we assume the implementation of any RBAC administrative command as non-interactive multi-party computations which proceeds as follows: when executing any RBAC command, the manager first carries out some local computation according to the command to produce some update messages for the file system and also

the users. After that, those update messages will be sent via secure channels. The users will update their local states accordingly upon receiving the update messages. The file system proceeds the update in a similar manner.

The global state of a cRBAC system st_G at any point during its execution is given by the tuple $(st_M, fs, \{st_u\}_{u \in U})$, where st_M is the local state of the manager, fs is the state of the file system and st_u is the local state of each user u. Since the manager is tasked with access control policy administration, we assume the RBAC policy $S = (U, O, P, UA, PA)$ is a part of its local state st_M and let $\phi(st_G)$ denote the RBAC policy of the global state st_G.

3.3 Cryptographic RBAC Scheme

A cRBAC system is defined by a cRBAC scheme which consists of the following algorithms:

- Init, the *initialisation* algorithm: A probabilistic algorithm that takes the security parameter λ and a set of roles R as input and outputs the initial global state of the cRBAC system.

- AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, the *RBAC administrative* algorithms: Probabilistic algorithms that implement the corresponding RBAC administrative commands. Each of these algorithms takes the state of the manager st_M, the current state of the file system fs and the argument for the RBAC command arg as input and then outputs the updated state for the manager and the file system, and also a set of update messages $\{msg_u\}_{u \in U}$ for all the users $u \in U$.

- Read, the *read* algorithm: A deterministic algorithm that allows a user retrieve the current content of a file. It takes the user's local state st_u, the current state of the file system fs and a file name o as input and outputs the current content of the file o if u has the read permission; if not, or if the file is empty, the algorithm returns $\perp$;

- Write, the *write* algorithm: A probabilistic algorithm that allows a user write content to a file. It takes a user's local state st_u , the current state of the file system fs, a file name o and the content to be written m as input and outputs the updated file system.

- Update, the *update* algorithm: A deterministic algorithm that takes the local state of a user st_u and an update message msg_u received from the manager and outputs the updated local state.

Recall that the write access to the file system is implemented by letting users append new versions to the file system. When one reads a file, it first needs to locate the latest valid file version so that it can retrieve the current content of the file. Therefore, in addition to the algorithms mentioned above, there also exists a sub-algorithm FindValidEntry.

- FindValidEntry: A deterministic algorithm that takes the local state of a user st_u, the current state of the file system *fs* and a file name *o* as input and outputs the most recent valid file version number. In the case that there is no valid version exists, it returns 0.

Notice that in a cRBAC system, the manager enforces the symbolic access control policy in a computational sense by generating appropriate cryptographic materials. It is therefore capable of retrieving the content from any file in order to carrying out some operations including file re-encryption. For simplicity, we let the manager retrieve the file content by running the user-specific algorithm Read with its local state st_M as input. Similarly, the manager can also locate the latest valid file version by running FindValidEntry on its own.

There is also a remark on the *updated file system*, which is as a part of the output of some algorithms outlined above. More specifically, the algorithms will produce update instructions to be carried out on the file system. For example, after running the Write algorithm, a user will get the update instruction *info* that includes the information of the file name and also the content to be appended to the file system. Then the user uploads *info* to the file system and the latter gets updated accordingly. The manager proceeds similarly but the update instructions might be different from the users due to its privilege of the data owner. For simplicity, we just let those algorithms output the updated file system. In terms of effect, all the above algorithms except Read can protentially update the global state of the cRBAC system. Therefore, we may write the execution of a cRBAC algorithm in the following form:

$$st_G \xrightarrow{Q} st'_G \Leftrightarrow st'_G \leftarrow\!\!\$\ Cmd(st_G, arg),$$

where *Cmd* is one of the algorithms that defines a cRBAC scheme, *arg* is its argument,

Q is a sequence of operations that implements the algorithm, st_G and st'_G are global state of the cRBAC system.

Let Q_i for $i = 0 \dots n$ be a sequence of operations, the execution of $\vec{Q} = (Q_0, ..., Q_n)$ can be written as:

$$st_{G_0} \xrightarrow{\vec{Q}} st_{G_{n+1}} \Leftrightarrow st_{G_0} \xrightarrow{Q_0} st_{G_1} \xrightarrow{Q_1}, ..., \xrightarrow{Q_{n-1}} st_{G_n} \xrightarrow{Q_n} st_{G_{n+1}},$$

where st_G and st'_G are global state of the cRBAC system.

Chapter 4

Game-Based Security of cRBAC

The contents presented in this chapter include results adapted from the paper *Policy Privacy in Cryptographic Access Control* [27] and also some new results from our follow-up work.

The aforementioned paper is a collaborative work with Anna Lisa Ferrara, Georg Fuchsbauer and Bogdan Warinschi. I am responsible for providing all the security definitions, the construction of the cRBAC system and the security statements with their proofs. My idea of policy privacy in access control systems is spurred by our work on using attribute-based signature schemes to enforce write access. Later, the idea is pushed further with Dr. Ferrara and Prof. Warinschi.

4.1 Introduction

The heavy reliance on reference monitors is a significant shortcoming of traditional access control mechanisms. It greatly impacts scalability and deployability, since monitors are single points of failure that need to run in protected mode and have to be permanently online to deal with every access request of users. Cryptography has the potential to alleviate this problem. This alternative approach that employs cryptographic primitives to enforce access control policies, is widely known as cryptographic access control. It aims to reduce the reliance on monitors or even eliminate this need, since the policy enforcement is achieved in an indirect way: data is protected by cryptographic primitives and the policies are enforced by distributing the appropriate keys to right users.

A primary concern of cryptographic access control is the large gap between the policy specification and the implementation of the access control system. It is best understood by contrasting it with policy enforcement via monitors. In monitor-based access con-

trol, every access request to the protected files is mediated by the reference monitor so that only the policy-compliant request will be granted. Therefore, the enforcement of access control policies holds by design. In cryptographic access control, the policy enforcement is more complicated. It relies on security guarantees of the underlying cryptographic primitives and also the appropriate key distribution/management in the system. Even though some advanced cryptographic primitives are seemingly well-suited for cryptographic access control, their security guarantees cannot be directly translated to security guarantee of the whole system. Therefore, formal security models for cryptographic access control systems are particularly important, since they establish the link between the implementation of the systems and the specification of access control policies, and also allow for rigorous security proofs.

There have been works in this area that focus on designing new primitives motivated by access control systems [32, 1, 42, 19, 67] and on designing access control systems based on those primitives [44, 66, 4, 70, 41]. Throughout the literature, rigorous definitions that look at the security of systems for access control have only been heuristically studied. Halevi et al. proposed a simulation-based security definition for access control on distributed file storage system in order to reason about the confinement problem [38]. Ferrara et al. defined a precise syntax for cryptographic role-based access control (cRBAC) systems and proposed a formal security definition with respect to secure read access [28]. They also suggested an construction based on predicate-encryption (PE). Their work eliminates the need for the trusted monitors to mediate every read access request, while write access is still delegated to the trusted monitors.

We follow the line of Ferrara et al.'s research on cRBAC systems and expand their works in several distinct directions. In Chapter 3, we have already introduced our extended system model that allows authorised users to execute write operations on files. Here, in this chapter, we are going to present our security definitions which aim to model the correct enforcement of the policies. More importantly, we initiate the study of policy privacy in the context of cryptographic access control. The information about the policy being enforced in a cryptographic access control system might be leaked during its execution. We begin to address this problem by providing formal security definitions to allow for rigorous study the information leak about the policy. We detail our contributions next.

4.1.1 Our results

Secure enforcement of RBAC policy. Our first results are formal security definitions for cRBAC systems. Very roughly, a cRBAC implementation is considered to be secure if it correctly enforces the RBAC policy. In order to formulate this, we propose multiple security definitions that capture the distinct security properties expected from a secure cRBAC system. Our security definitions are based on games, where the adversary is allowed to drive the execution of the system and to take over users. Then security is defined by measuring the inability of the adversary to triger some event during the execution or to distinguish between two possible executions.

The notions of *correctness* and *secure read access* were first introduced by Ferrara et al. in [28]. The former captures the security requirement that any user in the system should have access to the files to which it is allowed to access. The latter requires that by any means a user cannot learn any partial content of the file to which it does not have read access. Since their system model is extended here to support for enforcing access control over write operations to the file system, the two existing security definitions need to be redefined in our current system model. Next, we introduce a security definition for write access, which is called *secure write access*. Informally, it requires that all those contents written by unauthorised users will not be interpreted as valid.

After having formally defined security of cRBAC systems in game-based setting, as a step towards the goal for providing formal security definitions that precisely capture secure policy enforcement, we then turn to study security of cRBAC systems in simulation-based setting. In the follow-up work, we identify two different types of security concerns which are not captured by the existing security definitions. The first one corresponds to the ability of retrieving the previous contents in an unauthorised manner. More specifically, a user who is authorised to read a file might be able to retrieve the previous contents of that file, even it is not authorised to get access to those contents. The second one is related to secure write access to the files. A user who has the write permission of a file might be able to cause the other users fail in writing contents to that file. We propose two new security definitions *past confidentiality* and *local correctness* accordingly to capture the above mentioned security concerns .

Policy privacy in cryptographic access control. Our second contribution is bringing forth the policy privacy issues that appear in the context of cryptographic access control systems. The problem does not occur in monitor-based policy enforcement: when

interacting with the access control system, users can only learn if they have access to certain resources or not, while no other information will be leaked. But in cryptographic access control, dynamic changes to the files due to the administrative RBAC commands may reveal some information about the access control policy being enforced, especially in the case that the adversary has some partial knowledge on the access structure. One may imagine numerous situations where this information is sensitive. For example, in a paper submission system, one may always want to keep hidden the information about the PC members who have been assigned with certain papers in order to prevent authors from affecting them personally. In a hospital, it is always not desirable to leak if a patient's medical record can be accessed by certain specialists (e.g. oncologist and AIDS specialist, etc.).

Some existing works on cryptographic primitives tailored for access control have already attempted to deal with this type of leak. However, the privacy guarantees from the underlying cryptographic primitives may not suffice to preserve policy privacy in the access control systems that employ them. Moreover, there exists no security definition for policy privacy to allow one to formally prove that an implementation of the system can preserve such policy privacy guarantees.

Here, we clearly identify the abilities of an attacker and specify what are to be considered as privacy breaches in a cryptographic access control system. We propose multiple security definitions instead of a single one to allow for privacy-effiency trade-offs.

A construction of cRBAC. The additional security requirements mentioned above lead to a new construction of cRBAC system which strengthens the one proposed in the published paper [27]. We prove that our new construction meets the stronger security notions with respect to secure access, while offering a certain degree of privacy for the underlying policy.

To summarise, in this chapter we make the following contributions:

- We redefine two existing security definitions for cRBAC in the extended system model.
- We provide a formal security definition for *secure write access*.
- We provide two new security definitions called *past confidentiality* and *local correctness*.
- We provide formal security definitions for different flavours of *policy privacy*.
- We provide a construction which is proven to preserve correctness, write security, past confidentiality, local correctness and to preserve policy privacy to a certain extent.

4.2 Correctness

Informally, a cRBAC system is said to be *correct* if it guarantees that every user in the system can get access to the data for which it is authorised according to the policy. More specifically, a cRBAC system preserves correctness if:

1. any user u has the permission $(o, \texttt{read})$ should be able to retrieve the current content of o by reading it, and
2. the current content of a file o which is written by a user u who has the permission $(o, \texttt{write})$ will be correctly read by any other user who has the permission $(o, \texttt{read})$.

We formalise the requirements via a game between a challenger who acts as the manager of a cRBAC system and a polynomial-time adversary $\mathcal{A}$. The adversary is allowed to ask the manager to execute any RBAC administrative command and to request any user to write to the file system. But it cannot take over users. After carrying out some sequence of operations, $\mathcal{A}$ should show that the cRBAC system reaches the global state where there exists some user who cannot retrieve the current content of the file correctly to which he has the read access.

We define the following experiment $\mathbf{Exp}^{\text{corr}}_{\mathcal{CRBAC},\mathcal{A}}$. The experiment maintains the RBAC state of the system *State* which consists of (U, O, P, UA, PA). *State* is initialised as $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ and evolves according to the (symbolic) execution of the RBAC commands as requested by $\mathcal{A}$. It also maintains an object-indexed list T to record the latest content written to the files by authorised users. After the initialisation of the cRBAC

system with a given set of roles R, the adversary can call the oracles to execute the intended RBAC administrative commands and to write to the file system. Since secure channels are assumed and party corruption is not allowed, the adversary will be only provided the current state of the file system as the response for any query it makes. In addition, $\mathcal{A}$ is allowed to query the current state of the file system. At some point in the experiment, the adversary should output a user u^* along with a file o^*. The experiment terminates when $\mathcal{A}$ outputs a user-object pair (u^*, o^*). The adversary wins the game if u^* has the read permission of o^* but the content it retrieves from o^* by running $\mathsf{Read}(st_{u^*}, o^*, fs)$ does not match the record in $T[o^*]$.

Definition 7 (Correctness)**.** *A cRBAC system defined by the scheme* $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update) *is* ***correct*** *if for any probabilistic polynomial-time adversary* $\mathcal{A}$*, it holds that*

$$\mathbf{Adv}^{corr}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \Pr\left[\mathbf{Exp}^{corr}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \texttt{true}\right]$$

is 0*, where the experiment* $\mathbf{Exp}^{corr}_{\mathcal{CRBAC},\mathcal{A}}$ *is defined as follows:*

$\underline{\mathbf{Exp}^{corr}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$

$T \leftarrow \emptyset$; $State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

$(st_M, fs, \{st_u\}_{u \in U}) \leftarrow_{\$} \mathsf{Init}(1^\lambda, R)$

$(u^*, o^*) \leftarrow_{\$} \mathcal{A}(1^\lambda : \mathcal{O}_{corr})$

if $\mathsf{HasAccess}(u^*, (o^*, \mathsf{read})) \wedge T[o^*] \neq \mathsf{Read}(st_{u^*}, o^*, fs)$ **then**

  return `true`

else return `false`

The oracles $\mathcal{O}_{corr}$ that the adversary has access to are specified in Figure 4.1 and discussed below.

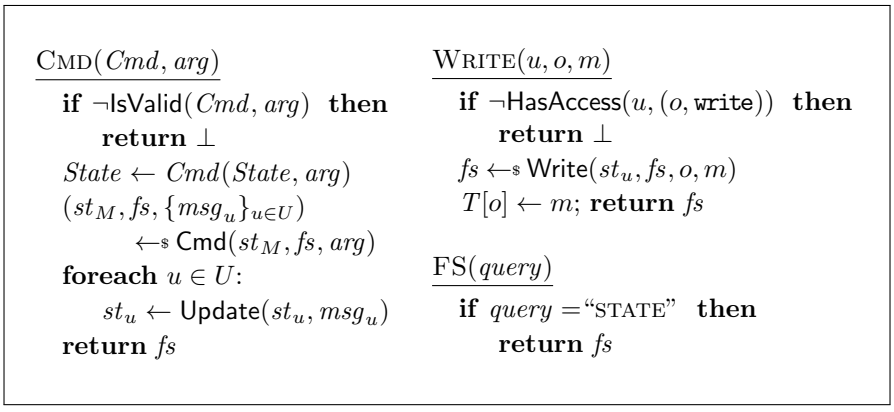

Figure 4.1: $\mathcal{O}_{corr}$: Oracles for defining the experiment $\mathbf{Exp}^{\text{corr}}_{\mathcal{CRBAC},\mathcal{A}}$.

The oracle CMD allows the adversary to ask the manager for the execution of any RBAC command by providing an RBAC administrative command *Cmd* (specified in 2.2) and the command-specific arguments *arg*. It will first check if the symbolic execution of *Cmd* with *arg* is valid: if not, an error is returned; otherwise, it will execute the command symbolically and then run the algorithm Cmd that implements the command. After that, $\mathcal{A}$ will be provided the current state of the file system.

The adversary can request an honest user u to write some content m to the file o. If u has the write permission of o, the oracle runs the algorithm Write to carry out the write opeartion and then stores m in $T[o]$. The adversary can check the current state of the file system at any point during the game by calling the oracle FS with the query "STATE" but appending file versions to the file system is not allowed here.

4.3 Read Security

In this section, we introduce two formal security definitions for a cRBAC system with respect to secure read access. The first one, called secure read access, is presented in the published paper [27]. The other security definition is called past confidentiality, which is strictly stronger than secure read access.

4.3.1 Secure Read Access

A cRBAC system is said to be secure with respect to read accesses if no user can deduce any partial content of a file without having the read permission. It is formalised via an indistinguishability-based game which involves a challenger who plays as the manager of a cRBAC system and an adversary $\mathcal{A}$. During the game, the adversary can choose two messages to be written to a file of which it does not have the read permission. Then one of the two messages will be written to that file and $\mathcal{A}$'s goal is to determine which of the messages it is.

More precisely, we define the following experiment $\mathbf{Exp}^{\text{read}}_{\mathcal{CRBAC},\mathcal{A}}$. The experiment starts with selecting a random bit $b \leftarrow\!\!\$\ \{0,1\}$. It maintains the symbolic RBAC state of the system as it evolves through the adversary's requests for the execution of RBAC commands. It also maintains a list *Cr* to record the corrupt users and another list *Ch* to record the files which are specified as challenges. The adversary can drive the execution of the system by asking the manager to execute any RBAC command and requesting any honest user to write to the file system. $\mathcal{A}$ can also take over any user by corrupting it and have unrestricted read and write (by appending) access to the file system. The

experiment only maintains local states for all honest users. For those corrupt users, their update messages will be sent to adversary instead. At some point, the adversary can ask for a challenge by specifying a tuple (u, o, m_0, m_1), where u is a user that has write access to the file o, m_0 and m_1 are two mesasges of the same length. In response, the challenger will run $\mathsf{Write}(st_u, o, m_b)$ to carry out the write operation on bahalf of u. When $\mathcal{A}$ terminates and outputs a guess of the random bit b', it wins the game if $b' = b$.

To prevent trivial wins, the experiment maintains the following invariant: there exists no user in the list Cr can have the read permission of any file in Ch, which means $\mathcal{A}$ cannot read the contents written to the challenge files directly by corrupting the users who are authorised to read.

Definition 8 (Secure Read Access)**.** *A cRBAC system which is defined by the scheme* $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update) *is* ***secure with respect to read access*** *if for any probabilistic polynomial-time adversary* $\mathcal{A}$*, it holds that*

$$\mathbf{Adv}^{read}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{read}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \mathtt{true}] - \tfrac{1}{2} \right|$$

is negligible in λ*, where* $\mathbf{Exp}^{read}_{\mathcal{CRBAC},\mathcal{A}}$ *is defined as follows:*

$\underline{\mathbf{Exp}^{read}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$

$b \leftarrow\!\!\$\ \{0,1\}$; $Cr, Ch \leftarrow \emptyset$

$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

$(st_M, fs, \{st_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$

$b' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \mathcal{O}_{read})$

return $(b' = b)$

The oracles $\mathcal{O}_{read}$ that the adversary has access to are specified in Figure 4.2 and discussed below.

Still, by calling the oracle Cmd the adversary can request for the execution of any administrative RBAC command, providing the symbolic execution of the command with its arguments is valid and it will not lead to a violation to the invariant. If the RBAC command causes some user or some file deleted from the system, the record in Cr or Ch will be removed accordingly.

The adversary can put a challenge by calling the oracle Challenge. Noticed that it is allowed to put multiple challenges in the game. In addition, Challenge returns an error if $\mathcal{A}$'s query will violate the invariant.

CMD(Cmd, arg)
if $\neg$IsValid(Cmd, arg) **then**
return $\perp$
$(U', O', P', UA', PA') \leftarrow Cmd(State, arg)$
foreach $u \in Cr$ AND $o \in Ch$:
if $\exists r \in R$:
$(u, r) \in UA' \wedge ((o, read), r) \in PA'$
then return $\perp$
$State \leftarrow (U', O', P', UA', PA')$
$(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow\!\!\!\!\!\!\$\ \mathsf{Cmd}(st_M, fs, arg)$
foreach $u \in Cr$:
if $u \notin U$ **then** $Cr \leftarrow Cr \setminus \{u\}$
foreach $o \in Ch$:
if $o \notin O$ **then** $Ch \leftarrow Ch \setminus \{o\}$
foreach $u \in U \setminus Cr$:
$st_u \leftarrow \mathsf{Update}(st_u, msg_u)$
return $(fs, \{msg_u\}_{u \in Cr})$

CORRUPTU(u)
if $u \notin U$ **then return** $\perp$
foreach $o \in Ch$:
if HasAccess$(u, (o, \mathtt{read}))$ **then**
return $\perp$
$Cr \leftarrow Cr \cup \{u\}$; **return** st_u

WRITE(u, o, m)
if $u \in Cr$ **then return** $\perp$
if $\neg$HasAccess$(u, (o, \mathtt{write}))$ **then**
return $\perp$
$fs \leftarrow\!\!\!\!\!\!\$\ \mathsf{Write}(st_u, fs, o, m)$
return fs

CHALLENGE(u, o, m_0, m_1)
if $\neg$HasAccess$(u, (o, \mathtt{write}))$ **then**
return $\perp$
if $|m_0| \neq |m_1|$ **then return** $\perp$
foreach $u' \in Cr$:
if HasAccess$(u', (o, \mathtt{read}))$ **then**
return $\perp$
$Ch \leftarrow Ch \cup \{o\}$
$fs \leftarrow\!\!\!\!\!\!\$\ \mathsf{Write}(st_u, fs, o, m_b)$
return fs

FS($query$)
if $query =$"STATE" **then**
return fs
if $query =$"APPEND($info$)" **then**
$fs \leftarrow fs \| info$; **return** fs

Figure 4.2: $\mathcal{O}_{read}$: Oracles for defining the experiment $\mathbf{Exp}^{\text{read}}_{\mathcal{CRBAC},\mathcal{A}}$.

The adversary can obtain the current state of the file system by calling the oracle FS with the query "STATE". To model the unrestricted append-only access to the file system, the adversary is allowed to write (append) arbitrary content to the file system by calling the oracle FS with the query "APPEND($info$)", where $info$ should contain a file name and the content to be appended to the file.

The experiment does not provide the adversary an oracle for read access to the file system. Since by corrupting users, the adversary can obtain their local states and receive user-specific update messages afterwards, which means $\mathcal{A}$ can retrieve file contents by running Read on its own.

4.3.2 Past Confidentiality

In our extended cRBAC system, the enforcement of access control on write access is supported by employing a versioning file storage where users can append contents only. The versioning file stroage allows users to have quasi-unrestricted read and write access to the file system, but it is also accompanied by some subtle security issues, even though it does not implement any access control mechanism.

The concept of file versions does not appear in traditional monitor-based access control. When a user gets access to the file to which it is authorised, only the current content will be available to it but not the previous contents. The previous contents here refer to those which are not a part of the current content. In cryptographic access control, due to the publicly accessible file system, users can easily obtain the previous file versions (even in an encrypted form) by monitoring the state of the file system. Therefore, a user who is recently granted the read permission of a file might have the ability to retrieve those previous contents which are written at the time when it does not have the permission - this can be considered as a violation of the access control being enforced.

The security concern mentioned above is not appropriately captured by the existing game-based security definitions of read security from the previous works [28, 27]. Recall that in those games that define the secure read access, the adversary is not allowed to get read access to the challenge files at any point during the game. This restriction imposed on the adversary leads to the attack mentioned above not being ruled out. In fact, the attack can be easily carried out in the constructions proposed in [28, 27]. Interestingly, the recently proposed constructions of cryptographic access control systems have the similar security concern [2, 43, 57], even though they have been proven to securely enforce the corresponding access control policies within their individual frameworks.

Here we propose a refinement of the existing definition of read security for cRBAC system. We name our strengthened security definition *past confidentiality*. The security property is formalised via the experiment $\mathbf{Exp}^{\mathrm{pc}}_{\mathcal{CRBAC},\mathcal{A}}$ which proceeds similarly to the $\mathbf{Exp}^{\mathrm{read}}_{\mathcal{CRBAC},\mathcal{A}}$, but the adversary here is allowed to corrupt the users who have the read permission of the challenge files under some conditions. The adversary's goal is still to determine a random bit $b \leftarrow\!\!\$\ \{0,1\}$ which is selected at the beginning of the game.

The experiment maintains the symbolic RBAC state of the system *State*, which is initalised as $(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset)$ and gets updated according to $\mathcal{A}$'s request for the execution of RBAC commands. The experiment keeps the following lists during the execution: *Cr* for the corrupt users, *Ch* for the files of which some contents have been specified as challenges, *L* for the users who have read access to the challenged contents and *Ud* for the files of which the current contents are specified as challenges.

In the experiment, the adversary can request for executing any RBAC administrative command, taking over users and requesting an honest user to write to a file with the content it specifies. The adversary can check the current state of the file system and

also write (append) some new content to it at any time during the experiment. $\mathcal{A}$ can ask for a challenge by specifying a tuple (u, o, m_0, m_1), where u is a user that has the write permission of the file o, m_0 and m_1 are two messages of the same length. Then the challenger will carry out $\mathsf{Write}(st_u, o, m_b)$ and provide the current state of the file system to the adversary as response. $\mathcal{A}$ can ask for multiple challenges. When $\mathcal{A}$ terminates with an output b', it wins the game if $b' = b$.

To prevent the adversary from winning the game trivially by corrupting a user who has read access to the challenge contents, the experiment maintains the following invariants. First, there exists no user in Cr can have read access to any file o in Ud. Second, no user in the list L can be corrupted. In other words, $\mathcal{A}$ is not allowed to grant the read permission of the challenge file to any corrupt user when the file's current content is specified as a challenge. Also, any user who has direct access to the challenge contents cannot be taken over by the adversary.

Definition 9 (Past Confidentiality)**.** *A cRBAC system defined by the scheme* $\mathcal{CRBAC} =$ (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update) *is said to preserve* ***past confidentiality*** *if for any probabilistic polynomial-time adversary* $\mathcal{A}$*, it holds that*

$$\mathbf{Adv}^{pc}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{pc}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \mathtt{true}] - \tfrac{1}{2} \right|$$

is negligible in λ*, where the experiment* $\mathbf{Exp}^{pc}_{\mathcal{CRBAC},\mathcal{A}}$ *is defined as follows:*

$\underline{\mathbf{Exp}^{pc}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$

$b \leftarrow\!\!\$\ \{0,1\}$; $Cr, Ch, L, Ud \leftarrow \emptyset$

$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

$(st_M, fs, \{st_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$

$b' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \mathcal{O}_{pc})$

return $(b' = b)$

The oracles $\mathcal{O}_{pc}$ that the adversary has access to are specified in Figure 4.3 and discussed below.

The oracle Cmd allows the adversary to request for the execution of any valid RBAC command. When $\mathcal{A}$'s query will lead to an update to Cr, Ch, L or Ud, the lists will get updated accordingly. When any user in L loses the read permission of any file in Ch, it will be removed from the list L. When $\mathcal{A}$ requests to grant the read permission of the files in Ud to an honest user, the user will be added to L.

CMD(Cmd, arg)
if $\neg$IsValid(Cmd, arg) **then**
return $\perp$
$(U', O', P', UA', PA') \leftarrow Cmd(State, arg)$
foreach $(u, o) \in Cr \times Ud$:
if $\exists r \in R$: $(u, r) \in UA'$
$\wedge\ ((o, \texttt{read}), r) \in PA'$ **then**
return $\perp$
$State \leftarrow (U', O', P', UA', PA')$
$(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Cmd}(st_M, fs, arg)$
foreach $u \in U \setminus L$:
if $\exists o \in Ud$: HasAccess$(u, (o, \texttt{read}))$ **then**
$L \leftarrow L \cup \{u\}$
foreach $u \in L$:
if $\nexists o \in Ch$: HasAccess$(u, (o, \texttt{read}))$
$\vee u \notin U$ **then**
$L \leftarrow L \setminus \{u\}$
foreach $o \in Ch$:
if $o \notin O$ **then**
$Ch \leftarrow Ch \setminus \{o\}$; $Ud \leftarrow Ud \setminus \{o\}$
foreach $u \in U \setminus Cr$:
$st_u \leftarrow \mathsf{Update}(st_u, msg_u)$
return $(fs, \{msg_u\}_{u \in Cr})$

CORRUPTU(u)
if $u \notin U \vee u \in L$ **then return** $\perp$
$Cr \leftarrow Cr \cup \{u\}$; **return** st_u

WRITE(u, o, m)
If $u \in Cr$ **then return** $\perp$
if $\neg$HasAccess$(u, (o, \texttt{write}))$ **then**
return $\perp$
$fs \leftarrow\!\!\$\ \mathsf{Write}(st_u, fs, o, m)$
if $o \in Ch$ **then**
$Ud \leftarrow Ud \setminus \{o\}$
return fs

CHALLENGE(u, o, m_0, m_1)
if $\neg$HasAccess$(u, (o, \texttt{write}))$ **then**
return $\perp$
if $|m_0| \neq |m_1|$ **then return** $\perp$
foreach $u' \in Cr$:
if HasAccess$(u', (o, \texttt{read}))$ **then**
return $\perp$
$fs \leftarrow\!\!\$\ \mathsf{Write}(st_u, fs, o, m_b)$
foreach $u' \in U$:
if HasAccess$(u', (o, \texttt{read}))$ **then**
$L \leftarrow L \cup \{u'\}$
$Ch \leftarrow Ch \cup \{o\}$; $Ud \leftarrow Ud \cup \{o\}$
return fs

FS($query$)
if $query =$"STATE" **then return** fs
if $query =$"APPEND($info$)" **then**
$fs \leftarrow fs \| info$; **return** fs

Figure 4.3: $\mathcal{O}_{pc}$: Oracles for defining the experiment $\mathbf{Exp}^{\text{pc}}_{\mathcal{CRBAC},\mathcal{A}}$.

When the adversary requests an honest user to write some content to a file of which the current content is specified as a challenge, the file will be removed from the list Ud, meaning from then on, the read permission of the file can be granted to a corrupt user.

When $\mathcal{A}$ requests to put a challenge by calling the oracle CHALLENGE, if there exists some corrupt user that has read access to the specified file, the oracle returns an error. Otherwise, it carries out the write operation and add the file to the lists Ch and Ud.

4.4 Write Security

The security definition for cRBAC system with respect to write security is first presented in our published paper [27]. When updating the paper for this thesis, we refine our security definition by making a small change to the adversary's output, which yields a slightly stronger security definition.

After presenting the security definition for write security, we will introduce a new security definition called local correctness. This security requirement is considered as a

sort of write security, but it is closely related to correctness.

4.4.1 Secure Write Access

We first introduce our security definition for secure write access. Informally, a cRBAC system is *secure* with respect to write accesses if no user can write any "valid" content to a file without having the write permission. Here "valid" means the entry appended by an unauthorised user is considered as valid and there is no requirement on wheather the content can be correctly retrieved or not. We use the term valid due to the use of open-accessible file system in our framework: every user can write to the file system by appending new file versions, but only those contents written (appended) by authorised users should be considered as valid.

We formalise this security requirement via a game between a challenger who plays the role as the manager of a cRBAC system and an adversary $\mathcal{A}$. The adversary can ask for executing any RBAC administrative command, impersonating any user by party corruption and writing some content to a file it specifies on behalf of any honest user. In addition, $\mathcal{A}$ is allowed to query the current state of the file system and also to append arbitrary content to it.

The experiment maintains the symbolic RBAC state of the system *State*, which is initialised as $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ and gets updated as the system evolves. It also keeps a list $Cr \in U$ to record the corrupt users and another object-indexed list T to record the contents written to the files by honest users. In addition, whenever there exists some corrupt user which has the write permission of o, $T[o]$ will store a special value `adv` and the content written by honest user will not be stored in $T[o]$. This remains invariant in the experiment.

When the adversary terminates with an output a file o^*, $\mathcal{A}$ wins the game if the content of o^* read by the manager is different from the record in $T[o^*]$ and $T[o^*] \neq$ `adv`, which means $\mathcal{A}$ has successfully written some valid content to o^* while no corrupt user can get write access to o^*, meaning it manages to write some content to the file successfully without having the permission.

To prevent trivial wins, from the point when the last write operation to the target file is carried out by an honest user till when $\mathcal{A}$ generates its output, no corrupt user is allowed to be granted the write permission of the target file. Otherwise, the adversary can write to the file system on his own with the use of the local states of any corrupt user who has write access to the target file.

Definition 10 (Secure Write Access)**.** *A cRBAC system which is defined by the scheme* $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update)*is* ***secure with respect to write access*** *if for any probabilistic polynomial-time adversary* $\mathcal{A}$*, it holds that*

$$\mathbf{Adv}^{write}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \Pr\left[\mathbf{Exp}^{write}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \texttt{true}\right]$$

is negligible in λ*, where* $\mathbf{Exp}^{write}_{\mathcal{CRBAC},\mathcal{A}}$ *is defined as follows:*

$\underline{\mathbf{Exp}^{write}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$

$Cr, T \leftarrow \emptyset$

$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

$(st_M, fs, \{st_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$

$o^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \mathcal{O}_{write})$

if $T[o^*] \neq \mathsf{adv} \wedge T[o^*] \neq \mathsf{Read}(st_M, fs, o^*)$ **then**

  return `true`

else return `false`

The oracles $\mathcal{O}_{write}$ that the adversary has access to are specified in Figure 4.4 and discussed below.

$\underline{\textsc{Cmd}(Cmd, arg)}$

if $\neg\mathsf{IsValid}(Cmd, arg)$ **then**
  return $\perp$
$State \leftarrow Cmd(State, arg)$
$(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Cmd}(st_M, fs, arg)$
if $Cmd =$ "DelObject" **then**
  Parse arg as o; $T[o] \leftarrow \emptyset$
if $Cmd =$ "DelUser" **then**
  Parse arg as u; $Cr \leftarrow Cr \setminus \{u\}$
foreach $o \in O$:
  if $\exists u' \in Cr : \mathsf{HasAccess}(u', (o, \texttt{write}))$ **then**
    $T[o] \leftarrow \mathsf{adv}$
foreach $u \in U \setminus Cr$:
  $st_u \leftarrow \mathsf{Update}(st_u, msg_u)$
return $(fs, \{msg_u\}_{u \in Cr})$

$\underline{\textsc{CorruptU}(u)}$

if $u \notin U$ **then return** $\perp$
foreach $o \in O$:
  if $\mathsf{HasAccess}(u, (o, \texttt{write}))$ **then**
    $T[o] \leftarrow \mathsf{adv}$
$Cr \leftarrow Cr \cup \{u\}$; **return** st_u

$\underline{\textsc{Write}(u, o, m)}$

if $u \in Cr$ **then return** $\perp$
if $\neg\mathsf{HasAccess}(u, (o, \texttt{write}))$ **then**
  return $\perp$
$fs \leftarrow\!\!\$\ \mathsf{Write}(st_u, fs, o, m)$
foreach $u' \in Cr$:
  if $\mathsf{HasAccess}(u', (o, \texttt{write}))$ **then**
    return fs
$T[o] \leftarrow m$; **return** fs

$\underline{\textsc{FS}(query)}$

if $query =$ "STATE" **then**
  return fs
if $query =$ "APPEND($info$)" **then**
  $fs \leftarrow fs \| info$; **return** fs

Figure 4.4: $\mathcal{O}_{write}$: Oracles for defining the experiment $\mathbf{Exp}^{\text{write}}_{\mathcal{CRBAC},\mathcal{A}}$.

By calling the oracle CMD, $\mathcal{A}$ can make the manager execute any RBAC command *Cmd* if its execution with the argument *arg* is valid. After executing the command symbolically, the oracle runs the corresponding algorithm Cmd to update the file system accordingly and to generate update messages for all the users. For every honest user, the oracle updates its local state by running Update with the dedicated update message. For those corrupt users, their update messages are sent to adversary. Throughout the game CMD ensures that whenever there is any corrupt user has the write permission of a file o, $T[o] = \texttt{adv}$.

The adversary can request for taking over any user $u \in U$ by calling the oracle CORRUPTU. The oracle then adds u to the list Cr and returns the local state st_u to $\mathcal{A}$. For every file $o \in O$ such that u has the permission $(o, \texttt{write})$, the record $T[o]$ will be assigned with the special value adv. The adversary can request an honest user to write some content m to a file o by calling the oracle WRITE. If the specified user has the permission, the oracle then runs Write with the user's local state to carry out the write operation. Only if there exists no user in the list Cr has the write permission of the file to be written to, $T[o]$ will store the content m. Otherwise, it stores the special value adv instead.

4.4.2 Local Correctness

The *local correctness* of a cRBAC system can be considered as a sort of write security notion. It captures the security concern from the "insiders" with respect to data availability. Namely, a user who has the write permission of a file should not be able to invalidate the file's future version which is written by an authorised user.

In other words, local correctness guarantees that even though there exists some corrupt user who has write access to a file, as long as it does not touch the file after an authorised user writes to the system, then any user who has the read permission should be able to retrieve the current content of that file.

This security requirement is formalised via the following experiment $\mathbf{Exp}^{\text{l-corr}}_{\mathcal{CRBAC},\mathcal{A}}$ that involves an adversary $\mathcal{A}$. The experiment maintains a list Cr to record the corrupt users and another object-indexed list T to record the contents written to files by the honest users. After the initialisation of the cRBAC system, the adversary can request for the execution of any administrative RBAC command, taking over any user and writing some content to a file on behalf of any honest user. $\mathcal{A}$ can also query for the current state of the file system and request to append arbitrary content to it.

The use of the list T here is different from that in the experiment of Definition 10. When an hoenst user writes some content to a file o, the content will be recorded in $T[o]$. If the adversary requests to update the file by appending any entry to it, $T[o]$ will store a special value `adv`, which means the file has been touched after the previous authorised write access.

The experiment terminates when the adversary outputs an object o^*. $\mathcal{A}$ wins the game if the content of o^* read by the manager is different from the record in $T[o^*]$ while $T[o^*]$ cannot be the special value `adv`.

Definition 11 (Local Correctness)**.** *A cRBAC system defined by the scheme* $\mathcal{CRBAC} =$ (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update) *is said to preserve* ***local correctness*** *if for any probabilistic polynomial-time adversary* $\mathcal{A}$*, it holds that*

$$\mathbf{Adv}^{l\text{-}corr}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \Pr\left[\mathbf{Exp}^{l\text{-}corr}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \texttt{true}\right]$$

is negligible in λ*, where* $\mathbf{Exp}^{l\text{-}corr}_{\mathcal{CRBAC},\mathcal{A}}$ *is defined as follows:*

$\underline{\mathbf{Exp}^{l\text{-}corr}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$

$T, Cr \leftarrow \emptyset$; $State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

$(st_M, fs, \{st_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$

$(u^*, o^*) \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \mathcal{O}_{l\text{-}corr})$

if $T[o^*] \neq \mathsf{adv} \wedge T[o^*] \neq \mathsf{Read}(st_{u^*}, o^*, fs)$ **then**

  return `true`

else return `false`

The oracles $\mathcal{O}_{l-corr}$ that the adversary has access to are specified in Figure 4.5.

To append some content to the file system, $\mathcal{A}$ can call FS with the query "APPEND(*info*)", where *info* should contain a file name o and the content to be appended to the file. Then $T[o]$ will store the special value `adv`.

4.5 Policy Privacy

In cryptographic access control systems, the particular type of privacy we are concerned with is related to the access polices. In traditional monitor-based access control, the users only know the resource they have access to and there is no other information is leaked. But when we use cryptographic techniques in access control, the information

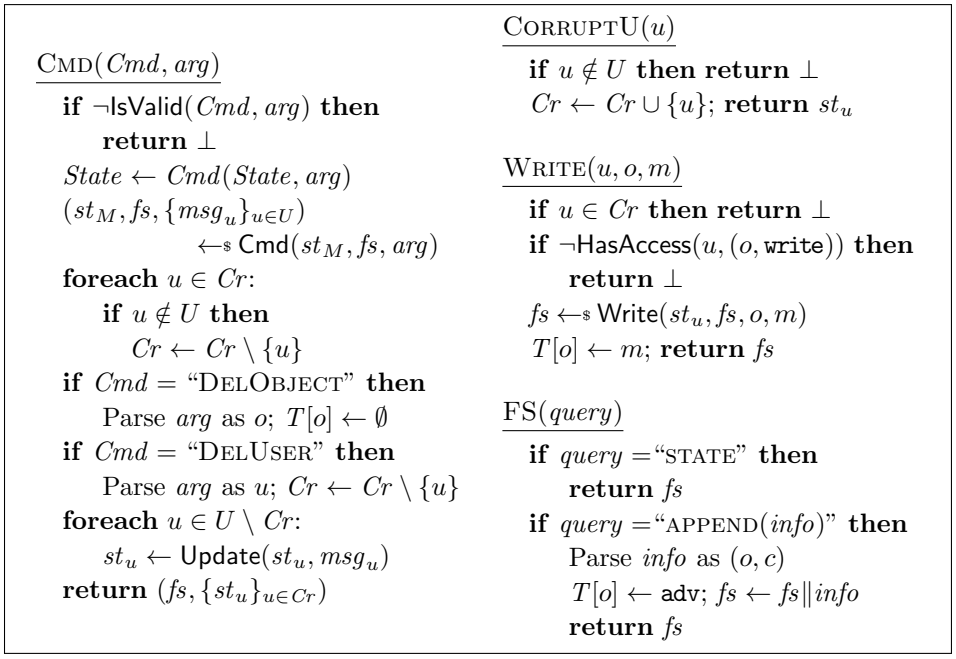

Figure 4.5: $\mathcal{O}_{l\text{-}corr}$: Oracles for defining the experiment $\mathbf{Exp}^{l\text{-}corr}_{CRBAC,\mathcal{A}}$.

about access policies might be unintentionally revealed, while such information might be sensitive.

In this section, we address the problem of keeping the access policy private by designing the formal security models that clearly identify the abilities of an attacker and specify what are to be considered as privacy breaches in cRBAC systems.

Here we are faced with a choice. One possibility is to provide a general privacy definition that any adversary would not be able to distinguish among any changes to the access control privacy. For example, we could require that no adversary can tell when a user is added to the system, or a permission has been revoked from some role. It is not difficult to see that such onerous requirements would immediately lead to prohibitively expensive implementations.

Instead, we pursue another approach where we identify privacy requirements seperately. This approach allows for trade-off between privacy and efficiency such that system designers can choose to sacrifice the privacy of some component deems less important in order to gain efficiency. The first distinction we made is to consider the privacy of the two matrices UA and PA separately. Then for each component we identify two further refinements. For privacy notions regarding the PA matrix we define two distinct notions: p2r-privacy, modelling the idea that a cRBAC system hides the assignment that maps a single permission to the roles that have it. Conversely, r2p-privacy demands that a cRBAC system hides the assignment that which permissions a certain role has. Similarly,

u2r-privacy and r2u-privacy model that a cRBAC system hides the assignment of a user to its roles and which users have a certain role.

We describe our formalisation of these notions below. For conciseness, we present one security experiment here and the different notions are obtained as instances.

The experiment involves a challenger who plays the role as the manager of a cRBAC system and an adversary $\mathcal{A}$. It starts with selecting a random bit $b \leftarrow\!\!\$\ \{0,1\}$. The experiment maintains the symbolic RBAC state of the system as it evolves throught adversary's requests for the execuiton of RBAC commands. The adversary can corrupt arbitrary users and can ask for performing a write operation on behalf of an honest user to some file with the content it specifies. Moreover, the adversary can query for the current state of the file system and also append information to it. At some point, the adversary can request for a challenge about privacy of the policy information in UA or in PA by calling the challenge oracle. The oracle can be called only once. For privacy related to PA, the adversary can speicify an RBAC command of either GrantPerm or RevokePerm with a quadruple $(p_0, p_1, r_0, r_1) \in P^2 \times R^2$. Then the manager will execute the command on (p_b, r_b) depending on the random bit b. After that, the adversary is not allowed to query oracles other than query the state of the file system and must output a guess of the bit b'. It wins the game if $b' = b$. The intuition behind the definition is that an adversary that observes the execution should not learn which of the two roles and which of the two permissions are involved in the execution of the command. We obtain two notions that capture different flavours of policy privacy related to the matrix PA by requiring $p_0 = p_1$, which defines p2r-privacy; and $r_0 = r_1$ which defines r2p-privacy. For instance, p2r-privacy models that a cRBAC system hides which roles have a certain permission by requiring the adversary not to be able to tell which of the two roles the permission has been granted/revoked.

We also define a weaker notion of p2r-privacy that we call p2r*-privacy. This notion is defined just like p2r-privacy except that the adversary can only request for permission granting when it queries the challenge oracle. Although very simple, p2r*-privacy is relevant for practical purposes. Indeed, in our motivating example of RBAC controlled access to hospital files, p2r*-privacy suffices to guarantee that granting access to the clinical record of a patient to a doctor would not reveal the speciality of the doctor.

Definition 12 (Policy Privacy)**.** *A cRBAC system defined by the scheme* $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update) *preserves x-privacy, where* $x \in \{\mathsf{u2r}, \mathsf{r2u}, \mathsf{p2r}, \mathsf{r2p}, \mathsf{p2r}^*\}$,

if for any probabilistic polynomial-time adversary $\mathcal{A}$, it holds that

$$\mathbf{Adv}^{x\text{-}privacy}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{x\text{-}privacy}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \mathtt{true}] - \tfrac{1}{2} \right|$$

is negligible in λ, where the experiment $\mathbf{Exp}^{x\text{-}privacy}_{\mathcal{CRBAC},\mathcal{A}}$ is defined as follows:

$\underline{\mathbf{Exp}^{x\text{-}privacy}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$

$b \leftarrow\!\!\$\ \{0,1\}$; $Cr \leftarrow \emptyset$

$(st_M, fs, \{st_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$

$b' \leftarrow \mathcal{A}(1^\lambda : \mathcal{O}_x)$

return $(b = b')$

The oracles $\mathcal{O}_x$ that the adversary has access to are specified in Figure 4.6. Here $\mathcal{O}_{\mathsf{u2r}}$ and $\mathcal{O}_{\mathsf{r2u}}$ consist of all oracles except ChllPA. Analogously, $\mathcal{O}_{\mathsf{p2r}}$, $\mathcal{O}_{\mathsf{r2p}}$ and $\mathcal{O}_{\mathsf{p2r}^*}$ are the oracles except ChllUA. The different notions are obtained via the restrictions outlined in the same figure. The adversary is allowed to call the challenge oracle only once, and after that it is not allowed to make any query to the oracles other than FS.

4.6 A Construction of cRBAC

In this section, we present our instantiation of cRBAC. We first describe how the files are stored in the file system and how to enforce access control on files via a combination of key-management and resigning/re-encrypting operations. Then we provide a detailed description of the algorithms of which our cRBAC scheme consists.

4.6.1 Overview of the Construction

The main ingredient of our cRBAC scheme is a PE-SK scheme. Specifically, we require that the PE-SK scheme is based on the Predicate Encryption for Non-Disjoint Sets (PE-NDS) scheme introduced in [28]. It is used as follows. To each role in the system we associate two attributes: attribute a_{rr} to which we refer as the read attribute of the role r and a_{rw} to which we refer as the write attribute of r. We use former to control reading rights associated to the role and latter to control writing rights. To all the users who have the role r, we provide them the decryption keys associated to a_{rr} and a_{rw} respectively. More precisely, each user in the system will be provided two keys and it will only hold two keys: one corresponds to the read attribtues of all the roles it belongs and the other corresponds to the write attributes of the same roles.

$\underline{\textsc{Cmd}(Cmd, arg)}$
if $\mathsf{challd} = 1$ **then return** $\perp$
if $\neg\mathsf{IsValid}(Cmd, arg)$ **then**
  return $\perp$
$State \leftarrow Cmd(State, arg)$
$(st_M, fs, \{msg_u\}_{u\in U}) \leftarrow_{\$} \mathsf{Cmd}(st_M, fs, arg)$
foreach $u \in U \setminus Cr$:
  $st_u \leftarrow \mathsf{Update}(st_u, msg_u)$
return $(fs, \{msg_u\}_{u\in Cr})$

$\underline{\textsc{CorruptU}(u)}$
if $\mathsf{challd} = 1$ **then return** $\perp$
if $u \notin U$ **then return** $\perp$
$Cr \leftarrow Cr \cup \{u\}$; **return** st_u

$\underline{\textsc{ChllUA}^{(x)}(Cmd, (u_0, u_1, r_0, r_1))}$
if $\mathsf{challd} = 1$ **then return** $\perp$
if $Cmd \notin \{AssignUser, DeassignUser\}$
    $\vee\neg\mathsf{IsValid}(Cmd, (u_0, r_0))$
    $\vee\neg\mathsf{IsValid}(Cmd, (u_1, r_1))$ **then**
  return $\perp$
if $(x = \mathsf{u2r} \wedge u_0 \neq u_1)$
    $\vee\ (x = \mathsf{r2u} \wedge r_0 \neq r_1)$ **then**
  return $\perp$
if $u_0 \in Cr \vee u_1 \in Cr$ **then return** $\perp$
$(st_M, fs, \{msg_u\}_{u\in U})$
    $\leftarrow_{\$} \mathsf{Cmd}(st_M, fs, (u_b, r_b))$
foreach $u \in U \setminus Cr$:
  $st_u \leftarrow \mathsf{Update}(st_u, msg_u)$
$\mathsf{challd} \leftarrow 1$; **return** $(fs, \{msg_u\}_{u\in Cr})$

$\underline{\textsc{Write}(u, o, m)}$
if $\mathsf{challd} = 1$ **then return** $\perp$
if $u \in Cr$ **then return** $\perp$
if $\neg\mathsf{HasAccess}(u, (o, \mathtt{write}))$ **then**
  return $\perp$
$fs \leftarrow_{\$} \mathsf{Write}(st_u, fs, o, m)$; **return** fs

$\underline{\textsc{FS}(query)}$
if $query =$"STATE" **then return** fs
if $query =$"APPEND($info$)" **then**
  $fs \leftarrow fs \| info$; **return** fs

$\underline{\textsc{ChllPA}^{(x)}(Cmd, (p_0, p_1, r_0, r_1))}$
if $\mathsf{challd} = 1$ **then return** $\perp$
if $Cmd \notin \{GrantPerm, RevokePerm\}$
    $\vee\neg\mathsf{IsValid}(Cmd, (p_0, r_0))$
    $\vee\neg\mathsf{IsValid}(Cmd, (p_1, r_1))$ **then**
  return $\perp$
if $(x = \mathsf{p2r} \wedge p_0 \neq p_1)$
    $\vee\ (x = \mathsf{r2p} \wedge r_0 \neq r_1)$
    $\vee\ (x = \mathsf{p2r}^* \wedge p_0 \neq p_1$
      $\wedge\ Cmd \neq GrantPerm)$ **then**
  return $\perp$
foreach $u \in Cr$:
  if $(u, r_0) \in UA \vee (u, r_1) \in UA$ **then**
    return $\perp$
$(st_M, fs, \{msg_u\}_{u\in U})$
    $\leftarrow_{\$} \mathsf{Cmd}(st_M, fs, (p_b, r_b))$
foreach $u \in U \setminus Cr$:
  $st_u \leftarrow \mathsf{Update}(st_u, msg_u)$
$\mathsf{challd} \leftarrow 1$; **return** $(fs, \{msg_u\}_{u\in Cr})$

Figure 4.6: $\mathcal{O}_x$: Oracles for defining the experiment $\mathbf{Exp}^{x\text{-}privacy}_{\mathcal{CRBAC},\mathcal{A}}$.

To control read access to a file o, we simply encrypt the file content under a public key that corredponds to *all* the read attributes of roles that have reading rights to o (computing such keys is one of the functionalities provided by PE-SK schemes). If a user is assigned with a role that has the reading right of o, it can retrieve the file content with the use of its decryption key for read access. To control write access, we use a standard digital signature scheme. Since all users in the system can append to the storage, the challenge is to ensure that only those contents appended by the users who have the writing right can be recognised as *valid*. We proceed as follows. To each file o, we associate it with a signing/verification key pair sk_o, vk_o. Users can update o by adding the modified variant to the storage, but only those updates that are signed with sk_o are valid, which can be verified by using vk_o. To ensure only the authorised users can obtain sk_o, we encrypt the signing key under *all* the attributes of the roles that

have write access to o and require that when writing to a file, a user needs to decrypt and retrieve the signing key first.

In more detail, we assume an append-only file system is (logically) organised as a matrix. Each row corresponding to a file and each column to a version of the file. The structure of a row in the file system is specified in Figure 4.7.

	Header	Versions			
Index	0	1	2	...	n
File	(pk; vk; sk)	(ctx_1,sig_1)	(ctx_2,sig_2)	...	(ctx_n,sig_n)

Figure 4.7: The structure of a row in the file system.

In the position $i = 0$ is the header of the file o to which the row corresponds. The information here is publicly readable but can be written only by the manager. It consists of three fields which, for a file o, we identify by $fs[o][0].pk$, $fs[o][0].vk$ and $fs[o][0].sk$. Here $fs[o][0].pk$ is the encryption key that corresponds to the read attributes of the roles that can read o, $fs[o][0].vk$ is the verification key associates to o and $fs[o][0].sk$ is the encryption of the signing key associates to o.

Users can append new versions to a file o by appending them to the row corresponding to o. Thus, each position $i > 0$ contains the i-th version of the file which we identify by $(fs[o][i].ctx, fs[o][i].sig)$. A valid entry on the row of the file o is of the form (ctx, sig), where ctx is the encryption of the file content and sig is a signature on ctx. In a normal execution, to add a new version to o an authorised user first needs to encrypt the file content under the public key $fs[o][0].pk$. Then it retrieves the signing key from $fs[o][0].sk$ and signs the encrypted content with the obtained signing key. To prevent the roll-back attack where a malicious user simply copies some old entries and appends them to the file, the signature is on the ciphertext together with the index that corresponds to the position of the row to which the new entry is appending. For a more powerful attack we called content-copying attack (will be explained in the next section), the content to be written to file needs to be appended with the index of the next available position and then gets encrypted under the public key associated to the file. The user then posts the ciphertext-signature pair to position i and this becomes the most recent version of the file.

Whenever an authorised user wishes to read a file o, she first needs to fetches the latest version of the file $(fs[o][i].ctx, fs[o][i].sig)$ for some $i > 0$, and determines whether $fs[o][i].sig$ is a valid signature and whether the signed index is equal to i. If not, she

fetches a previous version until a valid entry is encountered. When a valid entry is located at the position i of the file, the user decrypts $fs[o][i].ctx$ using her decryption key for read access and obtains some file content m which is concatenated with an index i'. In the case that $i' = i$, the content m is considered as the current content of o.

To add a new version to o, an authorised user first encrypts the new content under $fs[o][0].pk$, obtains sk_o by decrypting $fs[o][0].sk$ and uses it to sign the ciphertext. To prevent roll-back attacks where a malicious user simply copies some old entry, the signature is on the ciphertext together with the index of the entry. The user then appends the ciphertext-signature pair to the row corresponding to file o on the next empty position, and this becomes the most recent version of the file.

Whenever a role loses writing privileges to o, a new signature key pair for o is freshly generated. The new verification key is made public and the signing key is encrypted under the roles that still have the right to write. The latest valid version of the file is signed by the manager, so that the signature is valid under the new verification key associated to o.

Since multiple (encrypted) versions of the file are present in the system, the management of keys needs to be carefully crafted to avoid pitfalls where newly assigned rights permit access to old content. For example, whenever a read access is revoked from role r, the manager (1) assigns a fresh read attribute a_r to role r, (2) recomputes all the public keys for files to which r has still read access (to account for the changed attribute for r), (3) re-encrypts all latest (valid) ciphertexts under these public keys, and (4) sends the decryption keys associated to a_r to all users assigned to r.

Whenever a user is deassigned from a role r, the attribute for r is also changed and all the steps (1)–(4) are executed as above. In fact, when revoking a read permission p from some set of users, the local states of those users must be updated (will be demonstrated in Chapter 6). In addition, all signature key pairs for files to which r has write permission are also renewed; in particular, the new signing key is encrypted, and the concerned files will be re-signed to maintain validity under the new verification key. Thus if for example u is deassigned from r, she will no longer be able to decrypt the encrypted signing key and also the file contents using the old attributes associated to r.

4.6.2 $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ in details

Our cRBAC implementation $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ starts with the initialisation algorithm Init, which takes as input a security parameter and a set of roles. It first runs Setup, the setup

algorithm of $\mathcal{PE}$, with the security parameter and a sufficiently large attribute universe $A = \{1, \ldots, n_{max}\}$. It then initializes two role tables RT_{rd} and RT_{wr} with an increasing counter. Both tables are indexed by roles and they associate each role with two separate attributes (for read and write access respectively). Next, it initialises the file system fs, the symbolic RBAC state $State$ and two object-indexed list: SK for recording signing keys of objects in the file system and TT_{rd} for recording read attributes which helps the manager retrieve the contents of the files to which no user can get read access. Finally, it outputs the initial states of the manger st_M and the users st_u.

Algorithm $\mathsf{Init}(1^\lambda, R)$

$(mpk, mdk) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, A)$
$fs, RT_{\text{rd}}, RT_{\text{wr}}, TT_{\text{rd}}, SK \leftarrow \emptyset$; $ctr \leftarrow 1$
foreach $r \in R$:
  $RT_{\text{wr}}[r] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
foreach $r \in R$:
  $RT_{\text{wr}}[r] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
$st_M \leftarrow (mdk, RT_{\text{rd}}, RT_{\text{wr}}, TT_{\text{rd}}, SK, ctr, State)$
$fs[0][0] \leftarrow mpk$; $\{st_u\}_{u \in U} \leftarrow \emptyset$
return $(st_M, fs, \{st_u\}_{u \in U})$

Before we specify the algorithms implementing the RBAC commands, for convenience, we define the following auxiliary algorithms.

GetLength on input the state of the file system fs and a file o^* outputs the index of the last entry in $fs[o^*]$. EraseRest on input the state of the file system fs, a file o^* and an index idx erases all the entries at positions in $fs[o^*]$ greater than or equal to idx.

Algorithm $\mathsf{GetLength}(fs, o^*)$

if $o^* \notin O$ **then**
  return 0
for $i \leftarrow 1$ **to** ∞ :
  if $fs[o^*][i] = \emptyset$ **then**
    return $i - 1$

Algorithm $\mathsf{EraseRest}(fs, o^*, idx)$

if $o^* \notin O$ **then**
  return fs
for $i \leftarrow idx$ **to** $\mathsf{GetLength}(fs, o^*)$:
  $fs[o^*][i] \leftarrow \emptyset$
return fs

FindValidEntry on input of the state of the file system fs and a file o^* outputs the index i of the last entry that contains a valid signature. If there is no valid entry in o^*, FindValidEntry returns 0.

Algorithm $\mathsf{FindValidEntry}(fs, o^*)$

1 : **if** $o^* \notin O$ **then**
2 : **return** 0
3 : **for** $i \leftarrow \mathsf{GetLength}(fs, o^*)$ **to** 1 :
4 : $m \leftarrow fs[o^*][i].ctx \parallel i$
5 : **if** $\mathsf{Verify}(fs[o^*][0].vk, m, fs[o^*][i].sig) = 1$ **then**
6 : **return** i
7 : **return** 0

The following auxiliary algorithms are run by the manager only.

ReEnc on input the manager's state st_M, the state of the file system fs and a file o^* re-encrypts the content of its last valid entry according to the current encryption key of o^* and signs the new ciphertext using the signing key of o^*, which is stored in $SK[o^*]$. Then all the entries with the index greater than the last valid entry's will be erased.

The decryption of the content in the last valid entry of o^* is carried out with the use of a freshly generated decryption key by running DKGen on the predicate associates to the attributes of the roles that have the read permission of o^*. Here it is possible that there exist no such roles, this can be checked by looking up the record in $TT_{\mathrm{rd}}[o^*]$: in the case that $TT_{\mathrm{rd}}[o^*] \neq \emptyset$, it means the current content of o^* is written when there is no role has the read permission of o^*. Then a decryption key is generated with respect to the attribute stored in $TT_{\mathrm{rd}}[o^*]$. After decryption, the validity of the file content will be checked by comparing the obtained index i' with the index of the last valid entry of o^*, and comparing the obtained file name o' with the current file name o^*. If the file content is valid, the file content will be re-encryted and also a new signature will be generated with it.

Algorithm $\mathsf{ReEnc}(st_M, fs, o^*)$

1 : **if** $o^* \notin O$ **then**
2 : **return** fs
3 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
4 : **if** $i > 0$ **then**
5 : **if** $TT_{\mathrm{rd}}[o^*] \neq \emptyset$ **then**
6 : $x \leftarrow TT_{\mathrm{rd}}[o^*]$
7 : **else**
8 : $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \mathsf{read}), r) \in PA\}$
9 : $dk \leftarrow\!\!\$\ \mathsf{DKGen}(mdk, f_x)$
10 : $m \leftarrow \mathsf{Dec}(dk, fs[o^*][i].ctx)$
11 : **if** $m \neq \perp$ **then**
12 : Parse m as $m' \,\|\, i' \,\|\, o'$
13 : **if** $i' = i \wedge o' = o^*$ **then**
14 : $fs[o^*][i].ctx \leftarrow\!\!\$\ \mathsf{Enc}(fs[o^*][0].pk, m)$
15 : $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], fs[o^*][i].ctx \,\|\, i)$
16 : $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i+1)$
17 : **return** fs

ReSign on input the manager's state st_M, the state of the file system fs and a file o^* re-signs the last valid content of o^* using the signing key of o^*. Then all the entries with the index greater than the last valid entry of o^*'s will be erased.

Algorithm $\mathsf{ReSign}(st_M, fs, o^*)$

1 : **if** $o^* \notin O$ **then**
2 : **return** fs
3 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
4 : **if** $i > 0$ **then**
5 : $m \leftarrow fs[o^*][i].ctx \,\|\, i$
6 : $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], m)$
7 : $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i+1)$
8 : **return** fs

RoleUpdate takes as input the manager's state st_M, the state of the file system fs and a role r^* then assigns r^* with a new *read* attribute which is recorded in RT_{rd}. After that, the public keys and the content of the files for which r^* has read access are updated to the new attribute. The users of role r^* are issued new decryption keys for read access.

Algorithm $\mathsf{RoleUpdate}(st_M, fs, r^*)$

1 : **if** $r^* \notin R$ **then**
2 : **return** $(st_M, fs, \{\emptyset\}_{u\in U})$
3 : $\{rdk_u\}_{u\in U} \leftarrow \emptyset$
4 : $RT_{\mathrm{rd}}[r^*] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
5 : **foreach** $((o, \mathtt{read}), r^*) \in PA$:
6 : $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o, \mathtt{read}), r) \in PA\}$
7 : // y must be a non-empty here
8 : $fs[o][0].pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
9 : $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(st_M, fs, o)$
10 : **foreach** $(u, r^*) \in UA$:
11 : $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u, r) \in UA\}$
12 : $rdk_u \leftarrow\!\!\$\ \mathsf{DKGen}(mdk, f_x)$
13 : **return** $(st_M, fs, \{(rdk_u, \emptyset)\}_{u\in U})$

Now, we describe the algorithms implementing the RBAC commands. The algorithms that implement the RBAC commands are of the form that takes as input the manager's state st_M, the state of the file system fs and some argument specified by the RBAC command. It outputs the udpated state of the manager st_M, the file system fs and a set of update messages $\{msg_u\}_{u\in U}$ for all users.

AddUser simply adds a new user u^* to U. The algorithm AddObject adds a new object o^* to O, and adds the related permissions to P. It generates a signature key pair (sk_{o^*}, vk_{o^*}) by running KeyGen. Then sk_{o^*} is stored in $SK[o^*]$ while vk_{o^*} is stored in o^*'s header. It also runs PKGen with a distinct attribute to generate a public key for o^*. Such an attribute is used only once for providing a public key to allow legal users can write to the object, even if there is no user can get read access to it. In addition, the attribute will not be recorded in either RT_{rd} or RT_{wr} but in TT_{rd}, which means the manager will never issue a user with an decryption key that contains this attribute but he can still decrypt any ciphertext encrypted under this public key by generating a decryption key for the read attribute recorded TT_{rd}. At this point, the encrypted signing key field in o^*'s header remains empty.

Algorithm $\mathsf{AddUser}(st_M, fs, u^*)$

1 : **if** $u \notin U$ **then**
2 : $\quad U \leftarrow U \cup \{u^*\}$
3 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$

Algorithm $\mathsf{AddObject}(st_M, fs, o^*)$

1 : **if** $o \notin O$ **then**
2 : $\quad O \leftarrow O \cup \{o^*\}$
3 : $\quad P \leftarrow P \cup \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
4 : $\quad y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
5 : $\quad fs[o^*][0].pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
6 : $\quad TT_{\mathrm{rd}}[o^*] \leftarrow y$
7 : $\quad (sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda)$
8 : $\quad fs[o^*][0].vk \leftarrow vk_{o^*}; SK[o^*] \leftarrow sk_{o^*}$
9 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$

The algorithm AssignUser adds a new pair (u^*, r^*) to UA. The role r^* will be assigned with a new read attribute by running the algorithm RoleUpdate. Then the user u^* is given two decryption keys rdk and wdk which are for the sets of attributes corresponding to u^*'s current roles (via RT_{rd} and RT_{wr} individually), while the other users who are assigned with r^* will be provided a new decryption key for read access.

Algorithm $\mathsf{AssignUser}(st_M, fs, u^*, r^*)$

1 : **if** $u^* \notin U \vee r^* \notin R \vee (u^*, r^*) \in UA$ **then**
2 : $\quad$ **return** $(st_M, fs, \{\emptyset\}_{u \in U})$
3 : $UA \leftarrow UA \cup \{(u^*, r^*)\}$
4 : $\{msg_u\}_{u \in U} \leftarrow \emptyset$
5 : $(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(st_M, fs, r^*)$
6 : $x \leftarrow \{RT_{\mathrm{wr}}[r] \mid (u^*, r) \in UA\}$
7 : $wdk \leftarrow\!\!\$\ \mathsf{DKGen}(mdk, f_x)$
8 : $msg_{u^*}.wdk \leftarrow wdk$
9 : **return** $(st_M, fs, \{msg_u\}_{u \in U})$

GrantPerm adds a pair (p^*, r^*) to PA. If p^* is a read permission of some o^*, it first replaces the encryption key for o^* by a new one for the set of attributes corresponding to the roles (now including r^*) that have read access to o^*. The content of o^* is then re-encrypted under this new encryption key. Now there exists at least a role r^* can have read access to o^*, then the attribute stored in $TT_{\mathrm{rd}}[o^*]$ can be removed (recall that when $TT_{\mathrm{rd}}[o^*] \neq \emptyset$ it means no role can get read access to o^*).

If p^* is for write access, the signing key sk_{o^*} is encrypted with a set of attributes of the roles that have the permission p^* currently. Then the encrypted signing key is stored in o^*'s header to replace the previous one.

Algorithm $\mathsf{GrantPerm}(st_M, fs, p^*, r^*)$

1 : **if** $p^* \notin P \vee r^* \notin R \vee (p^*, r^*) \in PA$ **then**
2 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$
3 : $PA \leftarrow PA \cup \{(p^*, r^*)\}$
4 : Parse p^* as $(o^*, mode)$
5 : **if** $mode = \texttt{read}$ **then**
6 : $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \texttt{read}), r) \in PA\}$
7 : $fs[o^*][0].pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
8 : $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(st_M, fs, o^*)$
9 : $TT_{\mathrm{rd}}[o^*] \leftarrow \emptyset$
10 : **if** $mode = \texttt{write}$ **then**
11 : $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o^*, \texttt{write}), r) \in PA\}$
12 : // set of attributes for roles with write access to o^*
13 : $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
14 : $fs[o^*][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, SK[o^*])$
15 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$

DeassignUser removes (r^*, u^*) from UA and assigns a new read attribute to r^* using RoleUpdate. This then updates the file system accordingly and issues new decryption keys to the users having the role r^*. It also runs KeyGen to generate new signature key pairs for all the files for which r^* has write access. The new signing keys are encrypted under the attributes of the roles that have write access to them and are stored in the corresponding files' headers.

Algorithm $\mathsf{DeassignUser}(st_M, fs, u^*, r^*)$

1 : **if** $(u^*, r^*) \notin UA$ **then**
2 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$
3 : $UA \leftarrow UA \setminus \{(u^*, r^*)\}$
4 : $\{msg_u\}_{u \in U} \leftarrow \emptyset$
5 : $(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(st_M, fs, r^*)\}$
6 : **foreach** $((o, \texttt{write}), r^*) \in PA$:
7 : $(sk_o, vk_o) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda)$
8 : $SK[o] \leftarrow sk_o; fs \leftarrow\!\!\$\ \mathsf{ReSign}(st_M, fs, o)$
9 : $fs[o][0].vk \leftarrow vk_o$
10 : $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o, \texttt{write}), r) \in PA\}$
11 : $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
12 : $fs[o][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, sk_o)$
13 : **return** $(st_M, fs, \{msg_u\}_{u \in U})$

To delete a user u^* from U, DelUser first deassigns u^* from any of his roles and then updates U.

Algorithm $\mathsf{DelUser}(st_M, fs, u^*)$

1 : **if** $u^* \notin U$ **then**
2 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$
3 : **foreach** $(u^*, r) \in UA$:
4 : $(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow\!\!\$\ \mathsf{DeassignUser}(st_M, fs, u^*, r)$
5 : $U \leftarrow U \setminus \{u^*\}$
6 : **return** $(st_M, fs, \{msg_u\}_{u \in U})$

RevokePerm removes (p^*, r^*) from PA. If p^* is a read permission for some file o^*, the encryption key of o^* is renewed and current content of o^* is re-encrypted by using ReEnc. If there is no role has the read access to o^*, a new read attribute will be assigned for r^* and is stored in $TT_{\mathrm{rd}}[o^*]$. Then the read attribute associated to r^* is updated by running RoleUpdate (this is done so that users being assigned r^* later cannot decrypt ciphertexts of o^* from before the revocation).

If p^* is a write permission of some file o^*, a new signature key pair is generated for o^* and the last valid entry of o^* is re-signed with the new signing key. Then the new signing key is encrypted with the attribute set of the roles that write permission for o^* and stored in o^*'s header. When there is no user has the permission p^*, the manager will be the only one who can get access to the signing key. But the other users can still verify the validity of the entries of o^* by using the verification key.

Algorithm $\mathsf{RevokePerm}(st_M, fs, p^*, r^*)$

1 : **if** $(p^*, r^*) \notin PA$ **then**
2 : **return** $(st_M, fs, \{\emptyset\}_{u \in U})$
3 : $PA \leftarrow PA \setminus \{(p^*, r^*)\}$
4 : $\{msg_u\}_{u \in U} \leftarrow \emptyset$
5 : Parse p^* as $(o^*, mode)$
6 : **if** $mode = \texttt{read}$ **then**
7 : $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \texttt{read}), r) \in PA\}$
8 : **if** $y = \emptyset$ **then**
9 : $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
10 : $fs[o^*][0].pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
11 : $fs \leftarrow \mathsf{ReEnc}(st_M, fs, o^*)$
12 : $(st_M, fs, \{msg_u\}_{u \in U}) \leftarrow \mathsf{RoleUpdate}(st_M, fs, r^*)$
13 : **if** $mode = \texttt{write}$ **then**
14 : $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda)$
15 : $SK[o^*] \leftarrow sk_{o^*}; fs \leftarrow\!\!\$\ \mathsf{ReSign}(st_M, fs, o^*)$
16 : $fs[o^*][0].vk \leftarrow vk_{o^*}$
17 : $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o^*, \texttt{write}), r) \in PA\}$
18 : $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
19 : $fs[o^*][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, sk_{o^*})$
20 : **return** $(st_M, fs, \{msg_u\}_{u \in U})$

To delete an object o^*, the manager revokes every permission granted to o^* and updates O and P accordingly. Then all the entries of o^* is erased. The records in $TT_{\mathrm{rd}}[o^*]$ and $SK[o^*]$ will also be deleted.

Algorithm DelObject(st_M, fs, o^*)

1 : **if** $o^* \notin O$ **then**
2 : **return** $(st_M, fs, \{\emptyset\}_{u\in U})$
3 : $\{msg_u\}_{u\in U} \leftarrow \emptyset$
4 : **foreach** $(p, r) \in PA$:
5 : **if** $p \in \{((o^*, \texttt{read}), (o^*, \texttt{write})\}$ **then**
6 : $(st_M, fs, \{msg_u\}_{u\in U}) \leftarrow$ RevokePerm(st_M, fs, p, r)
7 : $O \leftarrow O \setminus \{o^*\}$
8 : $P \leftarrow P \setminus \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
9 : $TT_{\mathrm{rd}}[o^*], SK[o^*] \leftarrow \emptyset$
10 : $fs \leftarrow$ EraseRest$(fs, o^*, 0)$
11 : **return** $(st_M, fs, \{msg_u\}_{u\in U})$

Finally, we define the algorithms run by users: Update, Read and Write. The algorithm Update allows the users to get their local states updated. The users can get read and write access to files by running Read and Write.

An update message contains two decryption keys: *rdk* for read access and *wdk* for write access. When a user receives such a message from the manager, it runs algorithm Update to update its local state with the new keys.

Algorithm Update(st_u, msg_u)

1 : Parse msg_u as (rdk, wdk)
2 : **if** $rdk \neq \emptyset$ **then**
3 : $st_u.rdk \leftarrow rdk$
4 : **if** $wdk \neq \emptyset$ **then**
5 : $st_u.wdk \leftarrow wdk$
6 : **return** st_u

To write some content m to a file, a user first encrypts the concatenation of m with the index of the next available position i under the encryption key associated to the file. Next, she uses *wdk*, her decryption key for write access, to obtain the signing key encrypted in the file's header. She then signs the ciphertext along with the current index of the entry and the file name then appends a new entry containing the ciphertext and the signature to the file.

Algorithm $\mathsf{Write}(st, fs, o^*, m)$

1 : **if** $o^* \notin O$ **then**
2 : **return** fs
3 : Parse st as (rdk, wdk)
4 : $sk_{o^*} \leftarrow \mathsf{Dec}(wdk, fs[o^*][0].sk)$
5 : $i \leftarrow \mathsf{GetLength}(fs, o^*)$
6 : $ctx \leftarrow\!\!\$\ \mathsf{Enc}(fs[o^*][0].pk, m \,\|\, i+1 \,\|\, o^*)$
7 : $sig \leftarrow\!\!\$\ \mathsf{Sign}(sk_{o^*}, ctx \,\|\, i+1)$
8 : $fs[o^*][i+1].ctx \leftarrow ctx$
9 : $fs[o^*][i+1].sig \leftarrow sig$
10 : **return** fs

To read a file, a user first needs to locate the last valid entry of the file by verifying the signatures of the entires starting from the last entry. Let i be the index of the first entry found to contain a valid signature. If $i = 0$, meaning the file has no valid entry, Read outputs $\emptyset$; otherwise, the user uses her decryption key for read access, to decrypt the ciphertext of the last valid entry to obtain some file content m' with an index i' and a file name o'. If $i' = i$ and $o' = o^*$, meaning m' is valid then Read outputs m'; otherwise, the algorithm outputs $\perp$.

Algorithm $\mathsf{Read}(st, fs, o^*)$

1 : **if** $o^* \notin O$ **then**
2 : **return** $\perp$
3 : Parse st as (rdk, wdk)
4 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
5 : **if** $i > 0$ **then**
6 : $m \leftarrow \mathsf{Dec}(rdk, fs[o^*][i].ctx)$
7 : **if** $m \neq \perp$ **then**
8 : Parse m as $m' \,\|\, i' \,\|\, o'$
9 : **if** $i' = i \wedge o' = o^*$ **then**
10 : **return** m'
11 : **return** $\perp$
12 : **return** $\emptyset$

4.6.3 Cost analysis of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$

We remark that the main contribution of this chapter is the rigorous security definitions for cRBAC systems with respect to secure access and policy privacy. The construction we propose are not efficient and should be regarded as a proof of concept showing that secure policy enforcement and also meaningful levels of policy privacy can be achieved. Nevertheless, we list the costs of the administrative RBAC operations and read/write operation in Figure 4.8. Since the computation overhead of each operation depends on

the instantiation of the PE-SK scheme and also the digital signature scheme employed in our construction, by a slight abuse of notation we represent the computations in terms of the algorithms and even the operations themselves.

For simplicity, we also define the following notations:

- $U(r)$: the set of users which have been assigned with the role r.
- $R(u)$: the set of roles to which the user u has been assigned.
- $R(p)$: the set of roles to which the permission p has been granted.
- $O_r(r)$: the set of objects of which the read permissions have been granted to the role r.
- $O_w(r)$: the set of objects of which the write permissions have been granted to the role r.
- $V(o)$: the number of the invalid file versions appended to the file o after its last valid version.

Algorithm	**Computation Overhead**
$\mathsf{AddUser}(u)$	None
$\mathsf{AddObject}(o)$	$\mathsf{PKGen} + \mathsf{KeyGen}$
$\mathsf{AssignUser}(u, r)$	$\lvert O_r(r)\rvert \cdot (\mathsf{PKGen} + \mathsf{Dec} + \mathsf{Enc} + \mathsf{Sign}) + \sum_{o \in O_r(r)} V(o) \cdot \mathsf{Verify} +$ $(\lvert O_r(r)\rvert + U(r) + 1) \cdot \mathsf{DKGen}$
$\mathsf{GrantPerm}((o, \mathtt{read}), r)$	$V(o) \cdot \mathsf{Verify} + \mathsf{DKGen} + \mathsf{PKGen} + \mathsf{Dec} + \mathsf{Enc} + \mathsf{Sign}$
$\mathsf{GrantPerm}((o, \mathtt{write}), r)$	$\mathsf{PKGen} + \mathsf{Enc}$
$\mathsf{DeassignUser}(u, r)$	$(\lvert O_r(r)\rvert + \lvert O_w(r)\rvert) \cdot (\mathsf{PKGen} + \mathsf{Enc} + \mathsf{Sign}) + (\lvert O_r(r)\rvert + U(r)) \cdot$ $\mathsf{DKGen} + \lvert O_w(r)\rvert \cdot \mathsf{KeyGen} \ + (\sum_{o \in O_r(r)} V(o) + \lvert O_w(r)\rvert) \cdot$ $\mathsf{Verify} + \lvert O_r(r)\rvert \cdot \mathsf{Dec}$
$\mathsf{DelUser}(u)$	$\sum_{r \in R(u)} \mathsf{DeassignUser}(u, r)$
$\mathsf{RevokePerm}((o, \mathtt{read}), r)$	$(\lvert O_r(r)\rvert + 1) \cdot (\mathsf{PKGen} + \mathsf{Dec} + \mathsf{Enc} + \mathsf{Sign}) + \lvert O_r(r) + U(r) + 1\rvert \cdot$ $\mathsf{DKGen} + (V(o) + \sum_{o' \in O_r(r)} V(o')) \cdot \mathsf{Verify}$
$\mathsf{RevokePerm}((o, \mathtt{write}), r)$	$\mathsf{KeyGen} + V(o) \cdot \mathsf{Verify} + \mathsf{Sign} + \mathsf{PKGen} + \mathsf{Enc}$
$\mathsf{DelObject}(o)$	$\sum_{r \in R((o,\cdot))} \mathsf{RevokePerm}((o, \cdot), r)$

Figure 4.8: Cost analysis for the algorithms of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$.

4.7 Security of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$

Having proposed formal security definitions for cRBAC systems and provided the construction $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$, we now turn to examine security properties of our contruction.

Past Confidentiality $\Longrightarrow$ Read Security. We start with the following theorem, which shows that secure read access is implied by past confidentiality. This implication is not surprising at first glance, as the adversary in the game that defines past confidentiality is obviously more powerful due to its ability of granting read acess of the challenged files to corrupt users.

Theorem 1. *Past confidentiality is strictly stronger than secure read access.*

Proof sketch. We first show that any cRBAC system which preserves past confidentiality is secure with respect to read access. This part of proof is straightforward, since the reduction from past confidentiality to read security is obvious. Given any adversary $\mathcal{A}$ against read security of a cRBAC system, an adversary $\mathcal{B}$ for past confidentiality can be easily constructed. $\mathcal{B}$ runs a local copy of $\mathcal{A}$ and then simulates to it the read security game with the use of its oracles. During the simulation, $\mathcal{B}$ does not maintain the global state of the cRBAC system, but it keeps the lists defined in the experiment for read security. $\mathcal{B}$ starts the simulation by providing $\mathcal{A}$ the initial state of the file system it received from its challenger. Next, $\mathcal{B}$ simply forwards $\mathcal{A}$'s query to its oracles and then answers $\mathcal{A}$ with the response obtained from its oracles. If $\mathcal{A}$'s query will violate the restrictions of the read security game, $\mathcal{B}$ just replies with an error and ignores the query. When $\mathcal{A}$ outputs a guess of the random bit, $\mathcal{B}$ outputs the same guess.

Clearly, $\mathcal{B}$ just provides a perfect simulation. The global states in $\mathcal{B}$'s game and the simulated game are identical. All $\mathcal{A}$'s oracle queries will not lead to a violation to the restricitons of the past confidentiality game, since any query from $\mathcal{A}$ which does not violate the restrictions of the read security game will not violate any of the past confidentiality game's. In addition, the simulation directly depends on the random bit chosen in $\mathcal{B}$'s game. Thus, $\mathcal{B}$ wins the game with the same probability as $\mathcal{A}$ wins the simluated game. Thus, any cRBAC system is not secure with respect to read access does not preserves past confidentiality.

In addition, the construction of cRBAC system proposed in [27] has been proven to be secure with respect to read access. But clearly it does not preserve past confidentiality since granting the read permission of any file to a user will allow the user get access to those previous contents which are encrypted under the same public key. Therefore, we can conclude that past confidentiality is strictly stronger than secure read access. □

We next show that our construction preserves correctness, local correctness, past confidentiality, write security and $\mathsf{p2r}^*$-privacy. For brevity, we only provide the proof ideas for the security statements about correctness and local correctness.

Theorem 2. *If both the predicate encryption scheme $\mathcal{PE}$ and the signature scheme Σ are correct, $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ is correct (Definition 7).*

Proof idea. Recall that in the security game that defines correctness, the execution of the cRBAC system is only considered in a setting that taking over users is not permitted and users can update the file system by honestly running the write algorithm. In such case, correctness of the system solely depends on if the system design can provide the appropriate key management and resigning/re-encrypting operations. Meanwhile, from the specification of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$, we can observe that the design clearly satisfies the above requirement. Then in such a normal execution, if at some point there exists a user who cannot correctly retrieve the content of a file of which she has the read permission, there are only two possible reasons: signature verification failed and/or decryption failed. The former will lead to the user fetching some entry other than the last valid one or even cannot find any valid entry; while the latter will prevent the user from retrieving the signing key or the file content. Thus, if there exists any user who can break correctness of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$, it either breaks correctness of either $\mathcal{PE}$ or Σ. □

Theorem 3. *If both the predicate encryption scheme $\mathcal{PE}$ and the signature scheme Σ are correct, $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ preserves local correctness.*

Proof idea. In the security game that defines local correctness, it is required that from the time when the last write operation to a file carried out by an honest user untill the adversary terminates with that file as its output, appending any content to that file is not allowed. Before that, the adversary is allowed to corrupt any user who has the write permission of that file and to append arbitrary content to it.

We need to show that in $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$, no matter what content the adversary writes to a file, after that, any content written by an authorised user to the file will be correctly retrieved by any user who has the read permission.

From the specification of the write algorithm Write, we can observe that the algorithm will come up with the new entry to be appended to the file to be written to. The content of the new entry is completely independent from any of the previous entries and it only depends on the index of the file's last entry and the metadata stored in the header of that file. Since the file system is assumed to preserve correct ordering of the file indices and the metadata can only be updated by the manager, these two factors will not be affected by corrupt users' behaviours to the file system. In such case, the other possibility is either the predicate encryption scheme or the signature scheme is not correct and the

manager therefore cannot correctly retrieve the content written to the target file. Then we can conclude that $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ preserves local correctness under the assumption that both $\mathcal{PE}$ and Σ are correct. □

Past Confidentiality. One might expect, if the underlying encryption scheme is secure while the key-management and data encryption/re-encryption operations are performed appropriately according to the policy updates, the cRBAC system should preserve past confidentiality. However, this obvious intuition is not always true. Some care is needed to be taken since the system enforces access control on write access with the use of the append-only file system. The unrestricted write access to the files by appending new versions may concern read security of a cRBAC system. Consider the following "content-copying" attack. A malicious user may simply copy the encrypted content from some previous entry of a file (or even the entry from some other file), then come up with a new valid entry that contains the encrypted content and appends it to another file of which it has the write permission. Later, after being granted the read permission, the user might be able to retrieve the content by just reading it. This attack can be a potential threat to the cryptographic access control systems where the read access and write access are implemented based on separate mechanisms.

In fact, such an attack is thwarted in our implementation by requring the file content to be encrypted with the current index of the entry and also the file name. But this is not reflected in our theorem and its proof below.

The following theorem states that our cRBAC implementation preserves past confidentiality, under the assumption that all write operations are carried out by the manager, namely the manager will come up with the users' local states and write to the file system on behalf of them. Recall that the earlier result from Ferrara et al,'s work [28] showed read security of their cRBAC implementation solely relies on security of the underlying encryption scheme. In their system model, write opeartions are performed by the manager. Technically, this means that the simulator always knows the contents written to the file system when constructing the reduction. But our system model allows users append file entries on their own and the simulator is not always able to retrieve those contents. An alternative solution is to allow the manager to carry out the computation over the encrypted content (e.g. to have a construction that jointly uses a public key homomorphic encryption scheme with the predicate encryption scheme). However, it does not mean the new construction is more secure than the current one while it clearly leads to a more complicated result. Here, our theorem serves as a separation from the

previous result under the similar assumption.

Theorem 4. *If the PE-SK scheme $\mathcal{PE}$ has message-hiding ciphertexts, then $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ preserves past confidentiality in the setting that the manager carries out all write operations on behalf on the users.*

Proof. We prove the theorem by showing a reduction from past confidentiality of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ to message-hiding of $\mathcal{PE}$ under the assumption that all write operations can only be performed by the manager on behalf of the users, namely no user can append new entries to the file system on its own. Given any adversary $\mathcal{A}$ for $\mathbf{Exp}^{\text{pc}}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}$, an adversary $\mathcal{B}$ for $\mathbf{Exp}^{\text{msg-hide}}_{\mathcal{PE}}$ can be constructed with the use of $\mathcal{A}$ as a subroutine such that:

$$\mathbf{Adv}^{\text{msg-hide}}_{\mathcal{PE},\mathcal{B}}(\lambda) = \mathbf{Adv}^{\text{pc}}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma],\mathcal{A}}(\lambda).$$

Recall that in $\mathbf{Exp}^{\text{msg-hide}}_{\mathcal{PE}}$, the adversary is provided the master public key mpk by its challenger and has access to the following oracles: Oracle PKGen, on input a set of attributes y, returns a public key pk_y for y. pk_y will be stored in a list PK and y will be stored in another list $\mathcal{I}$ at the same position. Oracle DKGen, on input a predicate f, return a decryption key dk_f for f and records dk_f in a list F. Finally, oracle LR, on input an index k and a pair of mesages (m_0, m_1) returns the encryption of m_b under the public $PK[k]$ and adds $\mathcal{I}[k]$ to a list Ch, here b is a random bit chosen by the challenger. The oracles will return an error when any query from the adversary will lead to $f(I) = 1$ holds for some $f \in F$ and $I \in Ch$. Notice that our construction is based on predicate encryption for non-disjoint sets (PE-NDS) where the predicate is associated to a set of attributes $x \subseteq A$ and for any set of attributes $y \subseteq A : f_x(y) = 1 \Leftrightarrow x \cap y \neq \emptyset$. Let X be the union of the attributes associated to all the predicates queried to DKGen and let Y be the union of all attributes under which the the challenges were encrypted. Then thoughtout the game, oracle queries will be answered only if the the following invariant is maintained: $X \cap Y = \emptyset$, meaning no queried key can decrypt a challenge ciphertext.

We now describe how $\mathcal{B}$ works. $\mathcal{B}$ simulates $\mathbf{Exp}^{\text{pc}}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}$ depending on the random bit chosen by its challenger (and unknown to $\mathcal{B}$) and proceeds as the challenger of the simulated game. It starts from initialising a $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ as specified by Init, with the exception that it does not run the setup algorithm Setup of $\mathcal{PE}$ but just stores the received master public key mpk in $fs[0][0]$. $\mathcal{B}$ maintains two extra lists during the simulation: MS, the "message system", which is indexed by objects and it stores the current contents of the file system. When $\mathcal{B}$ needs to re-encrypt the (non-challenge) file

contents, it can look them up from MS instead of decrypting the ciphertexts. Thus, with the use of MS there is no need to maintain the list TT_{rd}; PK, a list to record the public keys generated by PKGen so far, which corresponds to the list PK maintained by $\mathcal{B}$'s challenger. It is used along with a counter ctr', which is identical to the counter ctr in $\mathcal{B}$'s own game. Later, when $\mathcal{B}$ needs to call LR for the encryption under some public key, it can look up PK to obtain the corresponding index of the key. Then $\mathcal{B}$ runs $\mathcal{A}$ internally and answers to $\mathcal{A}$'s queries as follows.

In general, $\mathcal{B}$ follows the specification of the oracles in the experiment $\mathbf{Exp}^{\mathrm{pc}}_{cRBAC}$ with the use of the implementation of the cRBAC scheme specified by $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$. Notice that $\mathcal{B}$ does not hold a master decryption key mdk, it therefore does not create and maintain decryption keys for honest users. For those corrupt users, $\mathcal{B}$ queries its oracle DKGen to obtain the decryption keys when needed. When $\mathcal{A}$ asks for excution of any valid RBAC command, $\mathcal{B}$ checks if the execution will lead to any corrupt user getting read access to the files of which the current contents are specified as challenges (i.e. those objects recorded in Ud). If so, $\mathcal{B}$ refuses the request and returns an error; otherwise, $\mathcal{B}$ executes the algorithm implementing the command and updates the system RBAC state accordingly. When $\mathcal{B}$ needs to generate a public key for some file, it queries PKGen with the set of attributes associated to the roles which have the permission to that file. Here the attribute set should be of the same type as the permission. When $\mathcal{B}$ needs to create decryption keys for corrupt users, it calls DKGen with predicates related to the attribute sets of the same type (either read attributes or write attributes). Any request for corrupting the users in the list L will be refused.

When $\mathcal{A}$ requests a user u^* to write some content to a file o^*, $\mathcal{B}$ records the content in $MS[o^*]$. Since it is assumed that all write access to the file system can only be carried out by the manager, u^* is no longer required to be an honest user here. $\mathcal{B}$ then retrieves the signing key from $SK[o^*]$ and uses it to generate a new file version for o^* directly, without querying the decryption key for u^*. This has the same effect as that u^* runs Write with its local state to write the content to o^*. If o^* is specified as a challenge, $\mathcal{B}$ removes o^* from the list Ud, namely the current content o^* is no longer a challenge.

When $\mathcal{A}$ wants to be challenged on some file o^* by specifying a user u^* and two messages m_0, m_1 of equal length, $\mathcal{B}$ first checks whether no corrupt user has read access to o^* at this point. If this is the case, it means that $\mathcal{B}$ has not made any decryption-key query for the predicate associated to any attribute under which o^* will be encrypted. $\mathcal{B}$ then does the followings: add o^* to both Ch and Ud, and record (m_0, m_1) in $MS[o^*]$ for

further use of re-encryption. Look up the index k of $fs[o^*][0].pk$ in PK and query the left-right oracle LR on $(k, m_0 \| i+1, m_1 \| i+1)$ to obtain the ciphertext and append it to $fs[o^*]$ together with a corresponding signature, here i is the index of the lastest version of o^*. Then add all the users who can get read access to o^* to the list L.

For any $o \in Ud$, it must hold that $MS[o]$ stores a pair of plaintext. If re-encryption is later required for a challenge file, $\mathcal{B}$ looks up the two plaintexts from MS, appends them with the index of the current file version, and again sends them to LR with the index of the file's current public key in PK. The public key here should be a new one, since re-encryption is required only after a new public key is generated for the file. After that, $\mathcal{B}$ updates the entry with the ciphertext received from LR and generates a new signature for it. For all other files, the re-encryption is done by looking up the file content in MS and encrypting the content under the new key.

When $\mathcal{A}$ terminates with a guess of the random bit b', $\mathcal{B}$ forwards it as the output. We now argue that the simulation provided by $\mathcal{B}$ is perfect.

First, we show that the invariant $X \cap Y = \emptyset$ is maintained thoughtout the game. Assuming by contradiction that the invariant is violated, namely $X \cap Y \neq \emptyset$. Then there exists an attribtue a such that $a \in X \cap Y$. We denote the role associated to the attribute a in the cRBAC system by r_a. Whenever the role is assigned with a new attribute, it is not considered as r_a from then on.

Recall that, the set X is the union of the attributes correspond to all the predicates for which decryption keys were queried and $\mathcal{B}$ calls DKGen only when it needs to generates decryption keys for corrupt users. Then set Y contains all the attributes under which the challenges were encrypted. $\mathcal{B}$ queries LR for a ciphertext only when $\mathcal{A}$ specifies its challenge or re-encryption is required for the challenged contents. Since Y contains only read attributes (those have been ever stored in RT_{rd}), $a \in X \cap Y$ implies that the following two conditions must have ever been met in the simulated game:

$$\exists u \in Cr : (u, r_a) \in UA \tag{C1}$$

$$\exists o \in Ud : ((o, \texttt{read}), r_a) \in PA \tag{C2}$$

Then there are two possibilities here: both the conditions (C1) and (C2) are satisfied simultaneously at some point during the game, or they are satisfied one after another.

Consider the first case, if at some point the two conditions hold, we immediately have:

$$\exists u \in Cr, o \in Ud : \mathsf{HasAccess}(u, (o, \mathtt{read})),$$

meaning there exists a corrupt user which is authorised to read one of the challenged contents. Obviously, this will never occur during the simulation. $\mathcal{B}$ maintains the list L to record the users who have read access to any of the challenge contents and it guarantees that the invariant $L \cap Cr \neq \emptyset$ should always hold during the game (in order to prevent trivial wins). Thus, there cannot exist any corrupt user that has the read permission of any challenge content during the simulation, which means that two conditions will not be satisfied simultaneously.

Now consider the other possibility, the two conditions are met one after another. We show that no matter which of the conditions is satisfied first, the other one will never hold later in the game.

In the case that (C1) is satisfied first, it means that $\mathcal{B}$ calls DKGen to obtain the decryption key for some corrupt user u who is assigned with the role r_a before it asks for a challenge under a public key with respect to the attribute set which contains a. As we have already known that at no point in the game, the two conditions can be both satisfied. Thus, in order to meet (C2), $\mathcal{A}$ must make queries to remove u from Cr (by calling DelUser) or deassign u from the role r_a (by calling DeassignUser). But from the specification of the algorithms DelUser and DeassignUser, we can observe that either call to invalidate (C1) will lead to a new attribute to be assigned to the role r_a and therefore $\mathcal{B}$ will not be able to request for a challenge with respect to the public key which is related to the attribute a from then on, which means (C2) will never occur later.

Similarly, in the case that (C2) is satisfied first, there should exist an object $o \in Ud$ such that $((o, \mathtt{read}), r_a) \in PA$ holds. Therefore, $\mathcal{A}$ needs to remove either o from Ud or $((o, \mathtt{read}), r_a)$ from PA before it requests to corrupt some user who is assigned with the role r_a. In order to remove o from Ud, $\mathcal{A}$ can request a user to write some content to o (by calling Write) or delete the object (by calling DelObject). After some new content has been written to o, $\mathcal{A}$ can assign some corrupt user with r_a (by calling AssignUser). However, in such case a new attribute will be assigned to r_a then (C1) will never hold later. In addition, at the moment when o is specified as the challenge, the users who have read access to o will be recorded in the list L, including all the users who have the role r_a. Even though $o \notin Ud$ holds after some new content is written to

o, $\mathcal{A}$ cannot corrupt any of these users before their read permission to o is revoked (by calling DEASSIGNUSER or REVOKEPERM). Still, from the specification of the algorithms DeassignUser and RevokePerm, either deassigning some user from the role r_a or revoking $(o, \texttt{read})$ from r_a will lead to a new attribute assigned to the role r_a. In addition, without writing some content to o, deleting o or just removing $(o, \texttt{read})$ from PA will also have the same effect to the attribute associated to r_a. So, if (C2) is satisfied first, (C1) will not occur later during the simulation. Therefore we can conclude that such an attribute $a \in X \cap Y$ does not exist so $X \cap Y = \emptyset$ is always maintained.

Moreover, $\mathcal{B}$ is in charge of the signature scheme Σ and during the simulation it can always udpate the file system correctly with the user of MS and SK. Thus, $\mathcal{B}$ provides perfect simulation of $\mathbf{Exp}^{\text{pc}}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}$ where the bit b is the same as the one chosen by $\mathcal{B}$'s challenger. Thus we have:

$$\mathbf{Adv}^{\text{msg-hide}}_{\mathcal{PE},\mathcal{B}}(\lambda) = \mathbf{Adv}^{\text{pc}}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma],\mathcal{A}}(\lambda).$$

We detail the adversary $\mathcal{B}$ in the following.

Adversary $\mathcal{B}(mpk : \text{PKGEN}, \text{DKGEN}, \text{LR})$

$Cr, Ch, Ud, L, MS, PK \leftarrow \emptyset$
$fs, RT_{\text{rd}}, RT_{\text{wr}}, SK \leftarrow \emptyset; ctr, ctr' \leftarrow 1$
foreach $r \in R$:
  $RT_{\text{rd}}[r] \leftarrow ctr; ctr \leftarrow ctr + 1$
foreach $r \in R$:
  $RT_{\text{wr}}[r] \leftarrow ctr; ctr \leftarrow ctr + 1$
$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset); fs[0][0] \leftarrow mpk$
$b' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \widetilde{\mathcal{O}}_{pc})$
return b'

The auxiliary algorithms ReEnc, ReSign and RoleUpdate used in $\mathcal{B}$'s simulation are specified as follows. The other algorithms GetLength, EraseRest and FindValidEntry are identical to those specified in the the cRBAC construction.

Algorithm $\mathsf{ReEnc}(fs, o^*)$

$\mathsf{ReEnc}(fs, o^*)$:
- **if** $o^* \notin O$ **then**
  - **return** fs
- $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
- **if** $i > 0$ **then**
  - **if** $o^* \in Ud$ **then**
    - Parse $MS[o^*]$ as (m_0, m_1)
    - Let k be such that : $PK[k] = fs[o^*][0].pk$
    - **Query** : $fs[o^*][i].ctx \leftarrow \mathrm{LR}(k, m_0 \,\|\, i \,\|\, o^*, m_1 \,\|\, i \,\|\, o^*)$
  - **else**
    - $fs[o^*][i].ctx \leftarrow\!\!\$\ \mathsf{Enc}(fs[o^*][0].pk, MS[o^*] \,\|\, i \,\|\, o^*)$
    - $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], fs[o^*][i].ctx \ \|\ i)$
- $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i+1)$
- **return** fs

Algorithm $\mathsf{ReSign}(fs, o^*)$

$\mathsf{ReSign}(fs, o^*)$:
- **if** $o \notin O$ **then**
  - **return** fs
- $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
- **if** $i > 0$ **then**
  - $m \leftarrow fs[o^*][i].ctx \ \|\ i$
  - $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], m)$
  - $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i+1)$
- **return** fs

Algorithm $\mathsf{RoleUpdate}(fs, r^*)$

$\mathsf{RoleUpdate}(fs, r^*)$:
- $\{rdk_u\}_{u \in Cr} \leftarrow \emptyset$
- $RT_{\mathrm{rd}}[r^*] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
- **foreach** $((o, \mathtt{read}), r^*) \in PA$:
  - $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o, \mathtt{read}), r) \in PA\}$
  - // y must be non-empty here
  - **Query** : $fs[o][0].pk \leftarrow \mathrm{PKGEN}(y)$
  - $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(fs, o)$
  - $PK[ctr'] \leftarrow fs[o^*][0].pk$
  - $ctr' \leftarrow ctr' + 1$
- **foreach** $u \in Cr$:
  - **if** $(u, r^*) \in UA$ **then**
    - $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u, r) \in UA\}$
    - **Query** : $rdk_u \leftarrow\!\!\$\ \mathrm{DKGEN}(f_x)$
- **return** $(fs, \{(rdk_u, \emptyset)\}_{u \in Cr})$

$\mathcal{B}$ maintains the oracles that $\mathcal{A}$ has access to as specified in Figure 4.9 and 4.10. Recall that in the security game that defines past confidentiality of cRBAC systems, the

Oracle AddUser(u^*)
- **if** $u \in U$ **then return** $\perp$
- $U \leftarrow U \cup \{u^*\}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AddObject(o^*)
- **if** $o \in O$ **then return** $\perp$
- $O \leftarrow O \cup \{o^*\}$
- $P \leftarrow P \cup \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
- $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
- **Query** : $fs[o^*][0].pk \leftarrow \text{PKGen}(y)$
- $PK[ctr'] \leftarrow fs[o^*][0].pk; ctr' \leftarrow ctr' + 1$
- $(sk_{o^*}, vk_{o^*}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$
- $fs[o^*][0].vk \leftarrow vk_{o^*}; SK[o^*] \leftarrow sk_{o^*}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AssignUser(u^*, r^*)
- **if** $u^* \notin U \vee r^* \notin R \vee (u^*, r^*) \in UA$ **then return** $\perp$
- **if** $\exists o \in Ud : ((o, \texttt{read}), r^*) \in PA$ **then**
  - **if** $u^* \in Cr$ **then return** $\perp$
  - **else** $L \leftarrow L \cup \{u\}$
- $UA \leftarrow UA \cup \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_\$ \mathsf{RoleUpdate}(fs, r^*)$
- $x \leftarrow \{RT_{\text{wr}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $wdk \leftarrow \text{DKGen}(f_x)$
- $msg_{u^*}.wdk \leftarrow wdk$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle DeassignUser(u^*, r^*)
- **if** $(u^*, r^*) \notin UA$ **then return** $\perp$
- $UA \leftarrow UA \setminus \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_\$ \mathsf{RoleUpdate}(fs, r^*)$
- **foreach** $((o, \texttt{write}), r^*) \in PA$:
  - $(sk_o, vk_o) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$
  - $SK[o] \leftarrow sk_o; fs \leftarrow_\$ \mathsf{ReSign}(fs, o)$
  - $fs[o][0].vk \leftarrow vk_o$
  - $y \leftarrow \{RT_{\text{wr}}[r] \mid ((o, \texttt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \text{PKGen}(y)$
  - $fs[o][0].sk \leftarrow_\$ \mathsf{Enc}(pk, sk_o)$
  - $PK[ctr'] \leftarrow pk; ctr' \leftarrow ctr' + 1$
- **foreach** $u \in L$:
  - **if** $\nexists o \in Ch : \mathsf{HasAccess}(u, (o, \texttt{read}))$ **then** $L \leftarrow L \setminus \{u\}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle GrantPerm(p^*, r^*)
- **if** $p^* \notin P \vee r^* \notin R \vee (p^*, r^*) \in PA$ **then return** $\perp$
- Parse p^* as $(o^*, mode)$
- **if** $o^* \in Ud \wedge mode = \texttt{read}$ **then**
  - **if** $\exists u \in Cr : (u, r^*) \in UA$ **then**
  - **else foreach** $u \in U \setminus L$:
    - **if** $(u, r^*) \in UA$ **then** $L \leftarrow L \cup \{u\}$
- $PA \leftarrow PA \cup \{(p^*, r^*)\}$
- **if** $mode = \texttt{read}$ **then**
  - $y \leftarrow \{RT_{\text{rd}}[r] \mid ((o^*, \texttt{read}), r) \in PA\}$
  - **Query** : $fs[o^*][0].pk \leftarrow \text{PKGen}(y)$
  - $fs \leftarrow_\$ \mathsf{ReEnc}(fs, o^*)$
  - $PK[ctr'] \leftarrow fs[o^*][0].pk; ctr' \leftarrow ctr' + 1$
- **if** $mode = \texttt{write}$ **then**
  - $y \leftarrow \{RT_{\text{wr}}[r] \mid ((o^*, \texttt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \text{PKGen}(y)$
  - $fs[o^*][0].sk \leftarrow_\$ \mathsf{Enc}(pk, SK[o^*])$
  - $PK[ctr'] \leftarrow pk; ctr' \leftarrow ctr' + 1$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle RevokePerm(p^*, r^*)
- **if** $(p^*, r^*) \notin PA$ **then return** $\perp$
- $PA \leftarrow PA \setminus \{(p^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- Parse p^* as $(o^*, mode)$
- **if** $mode = \texttt{read}$ **then**
  - $y \leftarrow \{RT_{\text{rd}}[r] \mid ((o^*, \texttt{read}), r) \in PA\}$
  - **if** $y = \emptyset$ **then**
    - $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
  - **Query** : $fs[o^*][0].pk \leftarrow \text{PKGen}(y)$
  - $fs \leftarrow_\$ \mathsf{ReEnc}(fs, o^*)$
  - $PK[ctr'] \leftarrow fs[o^*][0].pk; ctr' \leftarrow ctr' + 1$
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_\$ \mathsf{RoleUpdate}(fs, r^*)$
- **if** $mode = \texttt{write}$ **then**
  - $(sk_{o^*}, vk_{o^*}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$
  - $SK[o^*] \leftarrow sk_{o^*}; fs \leftarrow_\$ \mathsf{ReSign}(fs, o^*)$
  - $fs[o^*][0].vk \leftarrow vk_{o^*}$
  - $y \leftarrow \{RT_{\text{wr}}[r] \mid ((o^*, \texttt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \text{PKGen}(y)$
  - $fs[o^*][0].sk \leftarrow_\$ \mathsf{Enc}(pk, sk_{o^*})$
  - $PK[ctr'] \leftarrow pk; ctr' \leftarrow ctr' + 1$
- **foreach** $u \in L$:
  - **if** $\nexists o \in Ch : \mathsf{HasAccess}(u, (o, \texttt{read}))$ **then** $L \leftarrow L \setminus \{u\}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Figure 4.9: $\widetilde{\mathcal{O}}_{pc}$ (part 1)

Oracle DELOBJECT(o^*)
- **if** $o^* \notin O$ **then return** $\perp$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- **foreach** $(p, r) \in PA$:
  - **if** $p \in \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$ **then**
    - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$$ REVOKEPERM(p, r)
- $O \leftarrow O \setminus \{o^*\}$
- $P \leftarrow P \setminus \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
- $Ch \leftarrow Ch \setminus \{o^*\}$; $Ud \leftarrow Ud \setminus \{o^*\}$
- $MS[o^*], SK[o^*] \leftarrow \emptyset$
- $fs \leftarrow \mathsf{EraseRest}(fs, o^*, 0)$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle DELUSER(u^*)
- **if** $u^* \notin U$ **then return** $\perp$
- **foreach** $(u^*, r) \in UA$:
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$ \mathsf{DeassignUser}(fs, u^*, r)$
- $U \leftarrow U \setminus \{u^*\}$
- $Cr \leftarrow Cr \setminus \{u^*\}$; $L \leftarrow L \setminus \{u^*\}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle WRITE(u^*, o^*, m)
- **if** $\neg\mathsf{HasAccess}(u^*, (o, \texttt{write}))$ **then return** $\perp$
- $i \leftarrow \mathsf{GetLength}(fs, o^*)$
- $ctx \leftarrow\!\!\$ \mathsf{Enc}(fs[o^*][0].pk, m \,\|\, i+1 \,\|\, o^*)$
- $fs[o^*][i+1].ctx \leftarrow ctx$
- $sig \leftarrow\!\!\$ \mathsf{Sign}(SK[o^*], ctx \,\|\, i+1)$
- $fs[o^*][i+1].sig \leftarrow sig$
- $MS[o^*] \leftarrow m$; $Ud \leftarrow Ud \setminus \{o^*\}$
- **return** fs

Oracle CORRUPTU(u^*)
- **if** $u \notin U \vee u \in L$ **then return** $\perp$
- $Cr \leftarrow Cr \cup \{u^*\}$
- $x \leftarrow \{RT_{\mathrm{rd}}[r] \,|\, (u^*, r) \in UA\}$
- **Query** : $rdk \leftarrow \mathrm{DKGEN}(f_x)$
- $x' \leftarrow \{RT_{\mathrm{wr}}[r] \,|\, (u^*, r) \in UA\}$
- **Query** : $wdk \leftarrow \mathrm{DKGEN}(f_{x'})$
- **return** (rdk, wdk)

Oracle CHALLENGE(u^*, o^*, m_0, m_1)
- **if** $\neg\mathsf{HasAccess}(u^*, (o^*, \texttt{write}))$ **then return** $\perp$
- **if** $|m_0| \neq |m_1|$ **then return** $\perp$
- **foreach** $u \in Cr$:
  - **if** $\mathsf{HasAccess}(u, (o^*, \texttt{read}))$ **then return** $\perp$
- Let k be such that $PK[k] = fs[o^*][0].pk$
- **Query** : $ctx \leftarrow \mathrm{LR}(k, m_0 \,\|\, i+1, m_1 \,\|\, i+1)$
- $fs[o^*][i+1].ctx \leftarrow ctx$
- $sig \leftarrow\!\!\$ \mathsf{Sign}(SK[o^*], ctx \,\|\, i+1)$
- $fs[o^*][i+1].sig \leftarrow sig$
- $MS[o^*] \leftarrow (m_0, m_1)$
- **foreach** $u' \in U$:
  - **if** $\mathsf{HasAccess}(u', (o^*, \texttt{read}))$ **then** $L \leftarrow L \cup \{u'\}$
- $Ch \leftarrow Ch \cup \{o^*\}$; $Ud \leftarrow Ud \cup \{o^*\}$
- **return** fs

FS($query$)
- **if** $query =$ "STATE" **then return** fs

Figure 4.10: $\widetilde{\mathcal{O}}_{pc}$ (part 2)

adversary is allowed to get access to a single oracle CMD to request for the execution of any administrative RBAC command. Here, for clarity, it is achieved in a different favour but still has the same effect: the adversary now gets access to a group of oracles of which each corresponds to a single RBAC command. All of the oracles listed below follow the description of the oracle CMD with some slight changes (e.g. the condition of removing a user from the list L is more specific here). In each oracle, the run of the corresponding algorithm are replaced by those specified in the construction $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$. □

Theorem 5. *If the digital signature scheme Σ is existentially unforgeable under chosen-message attacks and the PE-SK scheme $\mathcal{PE}$ is message-hiding, $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ is secure with respect to write access.*

Proof. We prove this theorem through a sequence of games. It starts from the original security game that defines write security of cRBAC systems, and ends with a game where the adversary can gain advantage from the digital signature scheme Σ only. The description of the games and also the hops between successive games are as follows.

Game 0 : The initial game is simply the game that defines write security of cRBAC systems in the presence of an adversary $\mathcal{A}$. Recall that at the beginning of the game, the challenger initialises the system by running the initialisation algorithm Init with a set of roles. Then the adversary is given access to the oracles maintained by the challenger, by calling which it is allowed to request for the execution of any valid RBAC command, user corruption, writing on behalf of some honest user, querying the current state of the file system and appending data to the file system.

At some point during the game, the adversary terminates with an output of an object o^*. The advantage of the adversary in this game is defined by the probability that some content has been written to o^* in an unauthorised manner: the current content of o^* (read by challenger with the use of the manager's local state) differs from the last recorded written content.

Game 1 : We now transform *Game 0* into *Game 1* by requiring the adversary to write some valid content to the file associated to the verification and signing key pair specified by the challenger. More specifically, *Game 1* proceeds as the *Game 0* with the following changes.

At the beginning of the game, the challenger selects a random index i in the range of $\{1, \ldots, p(\lambda)\}$, where $p(\lambda)$ is a bound on the number of key pairs generated by running KeyGen, the key generation algorithm of Σ (if the adversary is polynomially bounded, this number is also polynomially bounded).

Whenever the challenger needs to generate a new verification and signing key pair for some file o, in case it is the i-th run of KeyGen, the challenger records the generated key pair (sk_i, vk_i) and stores the verification key vk_i in $fs[o][0].vk$. The signing key sk_i is encrypted with a set of attributes which are associated to the roles that have write access to o and the encrypted key is stored in $fs[o][0].sk$. When $\mathcal{A}$ outputs o^* and terminates, if the verification key stored in $fs[o^*][0].vk$ is not the recorded verification key, the challenger aborts the game.

Since the choice of the random index i is independent of the event that $\mathcal{A}$ manages

to write to the file o^*, it is clear that

$$\Pr[\mathsf{Game}_{\mathcal{A},1}] = \frac{1}{p(\lambda)} \Pr[\mathsf{Game}_{\mathcal{A},0}]. \tag{5.1}$$

Here and below we write for simplicity $\Pr[\mathsf{Game}_{\mathcal{A},i}]$ for the adventage of the adversary $\mathcal{A}$ in Game i.

Game 2 : This game proceeds as the one above, with the exception that in the i-th run of KeyGen, the challenger records the obtained key pair (vk_i, sk_i) and selects a random string of the same length of the signing key sk_i. Then it encrypts the random string and stores it in the related file's header instead of sk_i. Whenever the challenger needs to generate signatures for this file, it signs the messages with the recorded signing key until the key gets updated.

Lemma 1. *Let ϵ_0 be the advantage with which an efficient adversary can break message-hiding of the PE-SK scheme $\mathcal{PE}$, then:*

$$\left| \Pr[\mathsf{Game}_{\mathcal{A},1}] - \Pr[\mathsf{Game}_{\mathcal{A},2}] \right| = \epsilon_0.$$

We prove this lemma by constructing a distinguisher $\mathcal{D}$ given access to the oracles $\mathcal{O}$ = (PKGen, DKGen, LR), defined for message-hiding in Definition 6. The idea is, with use of these oracles $\mathcal{D}$ can simulate a hybrid game of *Game 1* and *Game 2* to an adversary $\mathcal{A}$. If there is a difference in the adversary's success probability between the two games, the distinguisher can gain the advantage equals to this in the message-hiding game of $\mathcal{PE}$.

In the i-th run of KeyGen for some file o, $\mathcal{D}$ checks if any corrupt user has write access to o. If so, $\mathcal{D}$ terminates and outputs 0. Recall that the winning condition of the game which requires that no corrupt user can have write access to the file outputs by $\mathcal{A}$ when $\mathcal{A}$ generates its output. It means in order to win the game, $\mathcal{A}$ needs to revoke the write permission from the corrupt users later and this will lead to the verification and signing key pair gets updated. Otherwise, $\mathcal{D}$ generates a key pair (sk_i, vk_i) and selects a random string rs of the same length as sk_i. $\mathcal{D}$ queries its LR oracle with (sk_i, rs) to obtain a ciphertext and stores it in $fs[o][0].sk$.

After that, when $\mathcal{D}$ needs to update $fs[o][0].sk$ due to $\mathcal{A}$'s oracle calls, it queries LR with the appropriate index and (sk_i, rs) to obtain the ciphertext. Meanwhile, if any of $\mathcal{A}$'s queries will lead to any corrut user can have the write permission of o or

$\mathcal{A}$ asks for executing some RBAC command which will also cause the current signing and verification key pair of o being updated, $\mathcal{D}$ terminates the simulation and outputs 0. Finally, when $\mathcal{A}$ outputs an object o^*, $\mathcal{D}$ outputs 1 if all the winning conditions are satisfied and else outputs 0. $\mathcal{D}$ is specified as follows.

Distinguisher $\mathcal{D}(mpk : \mathcal{O})$

$Cr, T, TT_{\mathrm{rd}}, SK, PK \leftarrow \emptyset$

$i \leftarrow\!\!\$\ \{1, ..., p(\lambda)\}; vk, sk, rs \leftarrow \emptyset$

$fs, RT_{\mathrm{rd}}, RT_{\mathrm{wr}} \leftarrow \emptyset; j, ctr, ctr' \leftarrow 1$

foreach $r \in R$:

  $RT_{\mathrm{rd}}[r] \leftarrow ctr; ctr \leftarrow ctr + 1$

foreach $r \in R$:

  $RT_{\mathrm{wr}}[r] \leftarrow ctr; ctr \leftarrow ctr + 1$

$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset); fs[0][0] \leftarrow mpk$

$o^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \widetilde{\mathcal{O}}_{write\text{-}1})$

if $fs[o^*].vk = vk \wedge T[o^*] \neq \mathsf{adv} \wedge T[o^*] \neq \mathsf{Read}(st_M, fs, o^*)$ **then**

  return 1

else return 0

The auxiliary algorithms of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ are also used in $\mathcal{D}$'s simulation, with some changes on the ReEnc, ReSign and RoleUpdate which are specified as follows.

Algorithm $\mathsf{ReEnc}(fs, o^*)$

1 : **if** $o^* \notin O$ **then**
2 : **return** fs
3 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
4 : **if** $i > 0$ **then**
5 : **if** $TT_{\mathrm{rd}}[o^*] \neq \emptyset$ **then**
6 : $x \leftarrow TT_{\mathrm{rd}}[o^*]$
7 : **else**
8 : $x \leftarrow \{RT_{\mathrm{rd}}[r] | ((o^*, \mathtt{read}), r) \in PA\}$
9 : **Query** : $dk \leftarrow \mathrm{DKGEN}(f_x)$
10 : $m \leftarrow \mathsf{Dec}(dk, fs[o^*][i].ctx)$
11 : $fs[o^*][i].ctx \leftarrow\!\!\$\ \mathsf{Enc}(fs[o^*][0].pk, m)$
12 : $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], fs[o^*][i].ctx \parallel i)$
13 : $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i + 1)$
14 : **return** fs

Algorithm $\mathsf{ReSign}(fs, o^*)$

1 : **if** $o \notin O$ **then**
2 : **return** fs
3 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
4 : **if** $i > 0$ **then**
5 : $m \leftarrow fs[o^*][i].ctx \parallel i$
6 : $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], m)$
7 : $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i+1)$
8 : **return** fs

Algorithm $\mathsf{RoleUpdate}(fs, r^*)$

1 : $\{rdk_u\}_{u \in Cr} \leftarrow \emptyset$
2 : $RT_{\mathrm{rd}}[r^*] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
3 : **foreach** $((o, \mathtt{read}), r^*) \in PA$:
4 : $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o, \mathtt{read}), r) \in PA\}$
5 : **Query** : $fs[o][0].pk \leftarrow \mathrm{PKGEN}(y)$
6 : $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(fs, o)$
7 : $PK[ctr'] \leftarrow fs[o^*][0].pk$
8 : $ctr' \leftarrow ctr' + 1$
9 : **foreach** $u \in Cr$:
10 : **if** $(u, r^*) \in UA$ **then**
11 : $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u, r) \in UA\}$
12 : **Query** : $rdk_u \leftarrow \mathrm{DKGEN}(f_x)$
13 : **return** $(fs, \{(rdk_u, \emptyset)\}_{u \in Cr})$

During the simulation, whenever $\mathcal{D}$ needs to run KeyGen to generate a verificaiton and signing key pair, it runs the following algorithm KeyGen' instead. KeyGen' takes as input the security parameter 1^λ and an object o^* and outputs the signature key pair by running KeyGen of the digital signature scheme. In the i-th run, if there exists some corrupt user who has the write permission of file that the new key pair will be associated to, $\mathcal{D}$ aborts the simulation and outputs 0; otherwise, it records the key pair in (sk, vk) and selects a random string rs of the same length as sk_{o^*}. In the case that the run of KeyGen' is to generate a new key pair for the object which is associated to (sk, vk), $\mathcal{D}$ also aborts and outputs 0.

Algorithm $\mathsf{KeyGen}'(1^\lambda, o^*)$

1 : **if** $fs[o^*][0].vk = vk$ **then**
2 : **output** 0
3 : $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda)$
4 : **if** $j = i$ **then**
5 : **if** $\exists u \in Cr : \mathsf{HasAccess}(u, (o^*, \mathtt{write}))$ **then**
6 : **output** 0
7 : $(sk, vk) \leftarrow (sk_{o^*}, vk_{o^*})$
8 : $rs \leftarrow\!\!\$\ \{0,1\}^{|sk_{o^*}|}$
9 : $j \leftarrow j + 1$
10 : **return** (sk_{o^*}, vk_{o^*})

The oracles that $\mathcal{D}$ maintains are specified in Figure 4.11 and 4.12. Again, $\mathcal{A}$ now can get access to a group of oracles correspond to the administrative RBAC commands which are functionally equivalent the oracle CMD in the security game that defines past confidentiality.

According to the specification of $\mathcal{D}$ above, it is clear that $\mathcal{D}$ will not lead to its oracles return an error. Since $\mathcal{D}$ calls LR only when it needs to generate the encryption of the signing key for the file associated to (vk_i, sk_i) and it ensures that no corrupt user can get write access to that file during the simulation. Therefore $\mathcal{D}$ will never request for any decryption key which allows it decrypt any of the ciphtertexts returned by LR. For those decryption keys that $\mathcal{D}$ queries for decrypting the file contents and the encrypted signing keys, they cannot be used to decrypt the challenge ciphertexts obtained from LR and therefore will not cause $\mathcal{D}$'s oracles return an error.

In the case that LR always returns the encryption of sk_i, meaning the random bit b selected in the message-hiding game is 0. Then the hybrid game is identical to *Game 1* and $\mathcal{D}$ outputs 1 with the same probability as $\mathcal{A}$'s advantage in *Game 1*. Then we have

$$\Pr[\mathcal{D}(mpk : \mathcal{O}) \rightarrow 1 \mid b = 0] = \Pr[\mathsf{Game}_{\mathcal{A},1}]. \tag{5.2}$$

Meanwhile, LR always returns the encryption of rs when $b = 1$, then the game is identical to *Game 2* and the probability that $\mathcal{D}$ outputs 1 is the same as $\mathcal{A}$'s advantage in *Game 2*. Thus we have

$$\Pr[\mathcal{D}(mpk : \mathcal{O}) \rightarrow 1 \mid b = 1] = \Pr[\mathsf{Game}_{\mathcal{A},2}], \tag{5.3}$$

Oracle AddUser(u^*)
- **if** $u \in U$ **then return** $\bot$
- $U \leftarrow U \cup \{u^*\}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AddObject(o^*)
- **if** $o \in O$ **then return** $\bot$
- $O \leftarrow O \cup \{o^*\}$
- $P \leftarrow P \cup \{(o^*, \mathtt{read}), (o^*, \mathtt{write})\}$
- $y \leftarrow \{ctr\}$; $ctr \leftarrow ctr + 1$
- **Query** : $fs[o^*][0].pk \leftarrow \text{PKGen}(y)$
- $PK[ctr'] \leftarrow fs[o^*][0].pk$; $ctr' \leftarrow ctr' + 1$
- $TT_{\text{rd}}[o^*] \leftarrow y$
- $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}'(1^\lambda, o^*)$
- $fs[o^*][0].vk \leftarrow vk_{o^*}$; $SK[o^*] \leftarrow sk_{o^*}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AssignUser(u^*, r^*)
- **if** $u^* \notin U \vee r^* \notin R \vee (u^*, r^*) \in UA$ **then return** $\bot$
- $UA \leftarrow UA \cup \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(fs, r^*)$
- $x \leftarrow \{RT_{\text{wr}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $wdk \leftarrow \text{DKGen}(f_x)$
- $msg_{u^*}.wdk \leftarrow wdk$
- **if** $u^* \in Cr$ **then**
  - **foreach** $((o, \mathtt{write}), r^*) \in PA$:
    - $T[o] \leftarrow \mathtt{adv}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle DeassignUser(u^*, r^*)
- **if** $(u^*, r^*) \notin UA$ **then return** $\bot$
- $UA \leftarrow UA \setminus \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(fs, r^*)$
- **foreach** $((o, \mathtt{write}), r^*) \in PA$:
  - $(sk_o, vk_o) \leftarrow\!\!\$\ \mathsf{KeyGen}'(1^\lambda, o)$
  - $SK[o] \leftarrow sk_o$; $fs \leftarrow\!\!\$\ \mathsf{ReSign}(fs, o)$
  - $fs[o][0].vk \leftarrow vk_o$
  - $y \leftarrow \{RT_{\text{wr}}[r] \mid ((o, \mathtt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \text{PKGen}(y)$
  - **if** $fs[o][0].vk = vk$ **then**
    - **Query** : $fs[o][0].sk \leftarrow \text{LR}(ctr', sk_o, rs)$
  - **else** $fs[o][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, sk_o)$
  - $PK[ctr'] \leftarrow pk$; $ctr' \leftarrow ctr' + 1$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle GrantPerm(p^*, r^*)
- **if** $p^* \notin P \vee r^* \notin R \vee (p^*, r^*) \in PA$ **then return** $\bot$
- Parse p^* as $(o^*, mode)$
- $PA \leftarrow PA \cup \{(p^*, r^*)\}$
- **if** $mode = \mathtt{read}$ **then**
  - $y \leftarrow \{RT_{\text{rd}}[r] \mid ((o^*, \mathtt{read}), r) \in PA\}$
  - **Query** : $fs[o^*][0].pk \leftarrow \text{PKGen}(y)$
  - $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(fs, o^*)$
  - $PK[ctr'] \leftarrow fs[o^*][0].pk$; $ctr' \leftarrow ctr' + 1$
  - $TT_{\text{rd}}[o^*] \leftarrow \emptyset$
- **if** $mode = \mathtt{write}$ **then**
  - $y \leftarrow \{RT_{\text{wr}}[r] \mid ((o^*, \mathtt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \text{PKGen}(y)$
  - **if** $fs[o][0].vk = vk$ **then**
    - **Query** : $fs[o^*][0].sk \leftarrow \text{LR}(ctr', SK[o^*], rs)$
  - **else** $fs[o^*][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, SK[o^*])$
  - $PK[ctr'] \leftarrow pk$; $ctr' \leftarrow ctr' + 1$
- **if** $\exists u \in Cr : \mathsf{HasAccess}(u, p^*)$ **then**
  - $T[o^*] \leftarrow \mathtt{adv}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle RevokePerm(p^*, r^*)
- **if** $(p^*, r^*) \notin PA$ **then return** $\bot$
- $PA \leftarrow PA \setminus \{(p^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- Parse p^* as $(o^*, mode)$
- **if** $mode = \mathtt{read}$ **then**
  - $y \leftarrow \{RT_{\text{rd}}[r] \mid ((o^*, \mathtt{read}), r) \in PA\}$
  - **if** $y = \emptyset$ **then**
    - $y \leftarrow \{ctr\}$; $ctr \leftarrow ctr + 1$
    - $TT_{\text{rd}}[o^*] \leftarrow y$
  - **Query** : $fs[o^*][0].pk \leftarrow \text{PKGen}(y)$
  - $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(fs, o^*)$
  - $PK[ctr'] \leftarrow fs[o^*][0].pk$; $ctr' \leftarrow ctr' + 1$
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(fs, r^*)$
- **if** $mode = \mathtt{write}$ **then**
  - $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}'(1^\lambda, o^*)$
  - $SK[o^*] \leftarrow sk_{o^*}$; $fs \leftarrow\!\!\$\ \mathsf{ReSign}(fs, o^*)$
  - $fs[o^*][0].vk \leftarrow vk_{o^*}$
  - $y \leftarrow \{RT_{\text{wr}}[r] \mid ((o^*, \mathtt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \text{PKGen}(y)$
  - **if** $fs[o^*][0].vk = vk$ **then**
    - **Query** : $fs[o^*][0].sk \leftarrow \text{LR}(ctr', sk_{o^*}, rs)$
  - **else** $fs[o^*][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, sk_{o^*})$
  - $PK[ctr'] \leftarrow pk$; $ctr' \leftarrow ctr' + 1$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Figure 4.11: $\widetilde{\mathcal{O}}_{\textit{write-1}}$ (part 1)

Oracle DELOBJECT(o^*)
- **if** $o^* \notin O$ **then return** $\perp$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- **foreach** $(p, r) \in PA$:
  - **if** $p \in \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$ **then**
    - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$$ REVOKEPERM(p, r)
- $O \leftarrow O \setminus \{o^*\}$
- $P \leftarrow P \setminus \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
- $T[o^*], SK[o^*] \leftarrow \emptyset$
- $fs \leftarrow \mathsf{EraseRest}(fs, o^*, 0)$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle WRITE(u^*, o^*, m)
- **if** $\neg\mathsf{HasAccess}(u^*, (o, \texttt{write}))$ **then**
  - **return** $\perp$
- $i \leftarrow \mathsf{GetLength}(fs, o^*)$
- $ctx \leftarrow\!\!\$\ \mathsf{Enc}(fs[o^*][0].pk, m \,\|\, i+1 \,\|\, o^*)$
- $fs[o^*][i+1].ctx \leftarrow ctx$
- $sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], ctx \,\|\, i+1 \,\|\, o^*)$
- $fs[o^*][i+1].sig \leftarrow sig$
- **foreach** $u \in Cr$:
  - **if** $\mathsf{HasAccess}(u, (o^*, \texttt{write}))$ **then**
    - **return** fs
- $T[o^*] \leftarrow m$; **return** fs

Oracle DELUSER(u^*)
- **if** $u^* \notin U$ **then return** $\perp$
- **foreach** $(u^*, r) \in UA$:
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{DeassignUser}(fs, u^*, r)$
- $U \leftarrow U \setminus \{u^*\}$; $Cr \leftarrow Cr \setminus \{u^*\}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle CORRUPTU(u^*)
- **if** $u \notin U$ **then return** $\perp$
- $Cr \leftarrow Cr \cup \{u^*\}$
- **foreach** $o \in O$:
  - **if** $\mathsf{HasAccess}(u^*, (o, \texttt{write}))$ **then**
    - $T[o] \leftarrow \texttt{adv}$
- $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $rdk \leftarrow$ DKGEN(f_x)
- $x' \leftarrow \{RT_{\mathrm{wr}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $wdk \leftarrow$ DKGEN$(f_{x'})$
- **return** (rdk, wdk)

FS($query$)
- **if** $query =$"STATE" **then**
  - **return** fs
- **if** $query =$"APPEND($info$)" **then**
  - $fs \leftarrow fs \| info$; **return** fs

Figure 4.12: $\widetilde{\mathcal{O}}_{write\text{-}1}$ (part 2)

Recall that, the advantage of $\mathcal{D}$ in the message-hiding game is

$$\left| \Pr[\mathcal{D}(mpk : \mathcal{O}) \to 1 \mid b = 0] - \Pr[\mathcal{D}(mpk : \mathcal{O}) \to 1 \mid b = 1] \right| = \epsilon_0. \tag{5.4}$$

Then combining Equations (5.2), (5.3) and (5.4), we have

$$\left| \Pr[\mathsf{Game}_{\mathcal{A},1}] - \Pr[\mathsf{Game}_{\mathcal{A},2}] \right| = \epsilon_0,$$

and the lemma is proved.

Lemma 2. *Let ϵ_1 be the advantage with which an efficient adversary gains in EUF-CMA attack game of the digital signature scheme Σ, then* $\Pr[\mathsf{Game}_{\mathcal{A},2}] = \epsilon_1$.

Assume that the file related to the verification and signing key pair (sk_i, vk_i), which is obtained from the i-th run of KeyGen, is o. We first observe that in *Game 2*, the encrypted "signing key" $fs[o][0].sk$ is independent of sk_i. Then $\mathcal{A}$ is provided the verification key vk_i and is allowed to see the signatures on messages chosen by itself. In this case, $\mathcal{A}$ wins the game only when he is able to forge a valid signature on his own. In

other words, from $\mathcal{A}$ we can construct an adversary $\mathcal{B}$ for $\mathbf{Exp}_{\Sigma}^{\text{eu-cma}}$ as follows. Given a verification key vk and the access to the oracle SIGN, $\mathcal{B}$ simulates for $\mathcal{A}$ *Game 2* by playing the role of the challenger. Here $\mathcal{B}$ is in charge of the PE-SK scheme $\mathcal{PE}$ and also the signature scheme Σ. It generates signing and verification key pairs for the files in its simulated game, except for the the one in the i-th run of KeyGen. By that time, $\mathcal{B}$ uses key vk that it receives instead of key vk_i and uses his signing oracle to produce the necessary signatures. When $\mathcal{A}$ terminates with an output o^*, $\mathcal{B}$ checks if all winning conditions are satisfied. If so, $\mathcal{B}$ outputs the file content and the signature of the last valid entry of o^*; otherwise, it aborts the simulation. The adversary $\mathcal{B}$ is detailed as follows.

Adversary $\mathcal{B}(vk : \text{SIGN})$

$(mpk, mdk) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, A)$

$Cr,\ T,\ TT_{\text{rd}}, SK \leftarrow \emptyset$

$i \leftarrow\!\!\$\ \{1, ..., p(\lambda)\}$

$fs, RT_{\text{rd}}, RT_{\text{wr}} \leftarrow \emptyset; j, ctr \leftarrow 1$

foreach $r \in R$:

  $RT_{\text{rd}}[r] \leftarrow ctr; ctr \leftarrow ctr + 1$

foreach $r \in R$:

  $RT_{\text{wr}}[r] \leftarrow ctr; ctr \leftarrow ctr + 1$

$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset); fs[0][0] \leftarrow mpk$

$st_M \leftarrow (mdk, RT_{\text{rd}}, RT_{\text{wr}}, TT_{\text{rd}}, SK, ctr, State)$

$o^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \widetilde{\mathcal{O}}_{write\text{-}2})$

if $fs[o^*].vk = vk \wedge T[o^*] \neq \mathsf{adv} \wedge T[o^*] \neq \mathsf{Read}(st_M, fs, o^*)$ **then**

  $idx \leftarrow \mathsf{FindValidEntry}(fs, o^*)$

  return $(fs[o^*][idx].ctx, fs[o^*][idx].sig)$

else abort

In $\mathcal{B}$'s simulation of *Game 2*, the auxiliary algorithms are identical to those specified in $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$. Still, $\mathcal{B}$ runs a modified algorithm KeyGen' instead of the key generation algorithm KeyGen of Σ. In the i-th run of KeyGen', it does not generate a signing and verification key pair. Instead, it returns vk, the verification key $\mathcal{B}$ obtains from its game, and a random string rs of the same length of the signing key sk which is unknown to $\mathcal{B}$.

Algorithm $\mathsf{KeyGen}'(1^\lambda, o^*)$

1 : **if** $fs[o^*][0].vk = vk$ **then**
2 : **output** 0
3 : **if** $j = i$ **then**
4 : **if** $\exists u \in Cr : \mathsf{HasAccess}(u, (o^*, \mathtt{write}))$ **then**
5 : **output** 0
6 : $rs \leftarrow\!\!\$\ \{0,1\}^{|sk|}; j \leftarrow j + 1$
7 : **return** (rs, vk)
8 : **else**
9 : $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda); j \leftarrow j + 1$
10 : **return** (sk_{o^*}, vk_{o^*})

The oracles that $\mathcal{D}$ maintains are specified in Figure 4.13 and 4.14. During the simulation, $\mathcal{D}$ is able to update the global state of the cRBAC system on its own since it is in charge of the PE-SK scheme and also the signature scheme. There is only one situation that $\mathcal{D}$ needs to call its own oracle SIGN, that is to provide signatures for the contents of the file associated to the verification key vk.

It is immediate that a successful attack of $\mathcal{A}$ against write security of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ translates into a forgery against the signature scheme Σ and we can conclude that

$$\Pr[\mathsf{Game}_{\mathcal{A},2}] = \epsilon_1,$$

where ϵ_1 is the advantage of $\mathcal{B}$ against Σ in the EU-CMA game. Thus the lemma is proved.

Now, combining Lemma 1 and 2, we have

$$\Pr[\mathsf{Game}_{\mathcal{A},1}] \leq \epsilon_0 + \epsilon_1. \tag{5.5}$$

From Equations (5.1) and (5.5), we can conclude that

$$\Pr[\mathsf{Game}_{\mathcal{A},0}] \leq (\epsilon_0 + \epsilon_1) \cdot p(\lambda).$$

Therefore, if the PE-SK scheme $\mathcal{PE}$ is message-hiding and the signature scheme Σ is existentially unforgeable under adaptive chosen-message attacks, then both ϵ_0 and ϵ_1 are negligible and therefore so is $\Pr[\mathsf{Game}_{\mathcal{A},0}]$. □

Theorem 6. *If the PE-SK scheme $\mathcal{PE}$ has attribute-hiding keys, $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ preserves* $\mathsf{p2r}^*$*-privacy.*

Proof. We prove this theorem by reducing $\mathsf{p2r}^*$-privacy of $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$ to identity-

Oracle AddUser(u^*)
- **if** $u \in U$ **then return** $\perp$
- $U \leftarrow U \cup \{u^*\}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AddObject(o^*)
- **if** $o \in O$ **then return** $\perp$
- $O \leftarrow O \cup \{o^*\}$
- $P \leftarrow P \cup \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
- $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
- $fs[o^*][0].pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
- $TT_{\mathrm{rd}}[o^*] \leftarrow y$
- $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}'(1^\lambda)$
- $fs[o^*][0].vk \leftarrow vk_{o^*}; SK[o^*] \leftarrow sk_{o^*}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AssignUser(u^*, r^*)
- **if** $u^* \notin U \vee r^* \notin R \vee (u^*, r^*) \in UA$ **then return** $\perp$
- $UA \leftarrow UA \cup \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(fs, r^*)$
- $x \leftarrow \{RT_{\mathrm{wr}}[r] \mid (u^*, r) \in UA\}$
- $wdk \leftarrow \mathsf{DKGen}(mdk, f_x)$
- $msg_{u^*}.wdk \leftarrow wdk$
- **if** $u^* \in Cr$ **then**
  - **foreach** $((o, \texttt{write}), r^*) \in PA$:
    - $T[o] \leftarrow \texttt{adv}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle DeassignUser(u^*, r^*)
- **if** $(u^*, r^*) \notin UA$ **then return** $\perp$
- $UA \leftarrow UA \setminus \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(fs, r^*)$
- **foreach** $((o, \texttt{write}), r^*) \in PA$:
  - $(sk_o, vk_o) \leftarrow\!\!\$\ \mathsf{KeyGen}'(1^\lambda, o)$
  - $SK[o] \leftarrow sk_o; fs \leftarrow\!\!\$\ \mathsf{ReSign}(fs, o)$
  - $fs[o][0].vk \leftarrow vk_o$
  - $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o, \texttt{write}), r) \in PA\}$
  - $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
  - $fs[o][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, sk_o)$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle GrantPerm(p^*, r^*)
- **if** $p^* \notin P \vee r^* \notin R \vee (p^*, r^*) \in PA$ **then return** $\perp$
- Parse p^* as $(o^*, mode)$
- $PA \leftarrow PA \cup \{(p^*, r^*)\}$
- **if** $mode = \texttt{read}$ **then**
  - $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \texttt{read}), r) \in PA\}$
  - $fs[o^*][0].pk \leftarrow \mathsf{PKGen}(mpk, y)$
  - $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(fs, o^*)$
  - $TT_{\mathrm{rd}}[o^*] \leftarrow \emptyset$
- **if** $mode = \texttt{write}$ **then**
  - $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o^*, \texttt{write}), r) \in PA\}$
  - $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
  - $fs[o^*][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, SK[o^*])$
- **if** $\exists u \in Cr : \mathsf{HasAccess}(u, p^*)$ **then**
  - $T[o^*] \leftarrow \texttt{adv}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle RevokePerm(p^*, r^*)
- **if** $(p^*, r^*) \notin PA$ **then return** $\perp$
- $PA \leftarrow PA \setminus \{(p^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- Parse p^* as $(o^*, mode)$
- **if** $mode = \texttt{read}$ **then**
  - $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \texttt{read}), r) \in PA\}$
  - **if** $y = \emptyset$ **then**
    - $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
    - $TT_{\mathrm{rd}}[o^*] \leftarrow y$
  - $fs[o^*][0].pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
  - $fs \leftarrow\!\!\$\ \mathsf{ReEnc}(fs, o^*)$
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow\!\!\$\ \mathsf{RoleUpdate}(fs, r^*)$
- **if** $mode = \texttt{write}$ **then**
  - $(sk_{o^*}, vk_{o^*}) \leftarrow\!\!\$\ \mathsf{KeyGen}'(1^\lambda, o^*)$
  - $SK[o^*] \leftarrow sk_{o^*}; fs \leftarrow\!\!\$\ \mathsf{ReSign}(fs, o^*)$
  - $fs[o^*][0].vk \leftarrow vk_{o^*}$
  - $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o^*, \texttt{write}), r) \in PA\}$
  - $pk \leftarrow\!\!\$\ \mathsf{PKGen}(mpk, y)$
  - $fs[o^*][0].sk \leftarrow\!\!\$\ \mathsf{Enc}(pk, sk_{o^*})$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Figure 4.13: $\widetilde{\mathcal{O}}_{write\text{-}2}$ (part 1)

hiding public keys of $\mathcal{PE}$. Let $\mathcal{A}$ be an adversary for $\mathbf{Exp}^{\mathsf{p2r}^*}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}$, we show that an adversary $\mathcal{B}$ for $\mathbf{Exp}^{id\text{-}h\text{-}pk}_{\mathcal{PE}}$ can be constructed with the use of $\mathcal{A}$ as a subroutine such that

$$\mathbf{Adv}^{id\text{-}h\text{-}pk}_{\mathcal{PE},\mathcal{B}}(\lambda) = \mathbf{Adv}^{\mathsf{p2r}^*-privacy}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma],\mathcal{A}}(\lambda).$$

Oracle DELOBJECT(o^*)
if $o^* \notin O$ **then return** $\perp$
$\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
foreach $(p, r) \in PA$:
  if $p \in \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$ **then**
    $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$}$ REVOKEPERM(p, r)
$O \leftarrow O \setminus \{o^*\}$
$P \leftarrow P \setminus \{(o^*, \texttt{read}), (o^*, \texttt{write})\}$
$T[o^*], SK[o^*] \leftarrow \emptyset$
$fs \leftarrow \mathsf{EraseRest}(fs, o^*, 0)$
return $(fs, \{msg_u\}_{u \in Cr})$

Oracle WRITE(u^*, o^*, m)
if $\neg\mathsf{HasAccess}(u^*, (o, \texttt{write}))$ **then**
  return $\perp$
$i \leftarrow \mathsf{GetLength}(fs, o^*)$
$ctx \leftarrow_{\$} \mathsf{Enc}(fs[o^*][0].pk, m \parallel i + 1)$
$fs[o^*][i + 1].ctx \leftarrow ctx$
if $fs[o^*][0].vk = vk$ **then**
  Query : $sig \leftarrow$ SIGN$(ctx \parallel i + 1)$
else $sig \leftarrow_{\$} \mathsf{Sign}(SK[o^*], ctx \parallel i + 1)$
$fs[o^*][i + 1].sig \leftarrow sig$
foreach $u \in Cr$:
  if $\mathsf{HasAccess}(u, (o^*, \texttt{write}))$ **then**
    return fs
$T[o^*] \leftarrow m$; **return** fs

Oracle DELUSER(u^*)
if $u^* \notin U$ **then return** $\perp$
foreach $(u^*, r) \in UA$:
  $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$} \mathsf{DeassignUser}(fs, u^*, r)$
$U \leftarrow U \setminus \{u^*\}$; $Cr \leftarrow Cr \setminus \{u^*\}$
return $(fs, \{msg_u\}_{u \in Cr})$

Oracle CORRUPTU(u^*)
if $u \notin U$ **then return** $\perp$
$Cr \leftarrow Cr \cup \{u^*\}$
foreach $o \in O$:
  if $\mathsf{HasAccess}(u^*, (o, \texttt{write}))$ **then**
    $T[o] \leftarrow \texttt{adv}$
$x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u^*, r) \in UA\}$
$rdk \leftarrow$ DKGEN(mdk, f_x)
$x' \leftarrow \{RT_{\mathrm{wr}}[r] \mid (u^*, r) \in UA\}$
$wdk \leftarrow \mathsf{DKGen}(mdk, f_{x'})$
return (rdk, wdk)

FS($query$)
if $query =$"STATE" **then**
  return fs
if $query =$"APPEND($info$)" **then**
  $fs \leftarrow fs \| info$; **return** fs

Figure 4.14: $\widetilde{\mathcal{O}}_{write\text{-}2}$ (part 2)

The idea is, with the access to the oracles of its own game, $\mathcal{B}$ can simulate to $\mathcal{A}$ the experiment $\mathbf{Exp}^{\mathsf{p2r}^*}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}$.

We now describe how $\mathcal{B}$ works. The simulation that $\mathcal{B}$ provides depends on the random bit b chosen by its own challenger. Upon receiving the master public key mpk from its challenger, $\mathcal{B}$ starts with the simulation of Init to initialise a $\mathcal{CRBAC}[\mathcal{PE}, \Sigma]$. Here $\mathcal{B}$ will not run the setup algorithm of $\mathcal{PE}$ but just stores mpk in $fs[0][0]$. It will generate the keys by calling its own oracles when needed. $\mathcal{B}$ then queries DKGEN with a predicate of the universe of the attribute $A = \{1, ..., n_{max}\}$ to obtain a decryption key dk_A. Notice that this key allows $\mathcal{B}$ access to all the files encrypted in the system, therefore $\mathcal{B}$ does not need to maintain the list TT_{rd} as specified in the cRBAC scheme. $\mathcal{B}$ then runs $\mathcal{A}$ internally and answers to its queries according to the specification of the oracles in $\mathbf{Exp}^{\mathsf{p2r}^*}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}$.

Whenever $\mathcal{B}$ needs to generate the public encryption key for some file, it submits to its own challenge oracle a pair of the same identities. When $\mathcal{A}$ asks to corrupt some

honest user, $\mathcal{B}$ calls DKGen with predicates related to the identities associated to the user's roles to obtain the two decryption keys for read and write access respectively then forwards them to $\mathcal{A}$. If $\mathcal{B}$ needs to decrypt some ciphertext of the system (e.g. to perform a re-encryption of some content), it uses dk_A to recover the plaintext.

Finally, when $\mathcal{A}$ queries the challenge oracle ChllPA with the query (p, p, r_0, r_1) (for $\mathsf{p2r}^*$-privacy, the specified command can only be $\mathsf{GrantPerm}$) where $p \in P$ and $r_0, r_1 \in R$, the adversary $\mathcal{B}$ checks that the query would have been valid in the identity-hiding public keys experiment. If this is not the case then it answers with an error $\perp$. Otherwise, $\mathcal{B}$ queries its own challenge oracle LR with (I_0, I_1) where $I_b = I \cup \{RT[r_b]\}$ with RT can be RT_{rd} or RT_{wr}, depending on the type of p and I is the set of attributes associated to the roles that have access to the permission p, which is retrieved from RT.

After $\mathcal{B}$ receives a response of a public key pk_b which is generated according the random bit chosen in $\mathcal{B}$'s game from LR, if the challenge permission is a read permission, $\mathcal{B}$ updates the corresponding file's public key with pk_b and re-encrypts the last valid entry of that file. If the permission is a write permission, $\mathcal{B}$ re-encrypts the signing key of the file with pk_b. Afterwards, all further oracle calls from $\mathcal{A}$ will be ignored.

Whenever $\mathcal{A}$ outputs a guess b', $\mathcal{B}$ outputs the same bit. We now argue that the simulation provided by $\mathcal{B}$ is perfect.

First notice that before $\mathcal{A}$ calls ChllPA, $\mathcal{B}$ is capable of coming up all the required cryptographic materials. Since $\mathcal{B}$ is fully in charge of the signature, it knows all of the signing keys and therefore can provide all the necessary signatures. Moreover, $\mathcal{B}$ is able to decrypt all the encrypted content of the file system, including signing keys stored in file headers and also the file contents. In addition, whenever $\mathcal{B}$ needs to generate public key for any file, it queries the oracle PKGen with the appropriate set of attributes to obtain one.

The details of the adversary $\mathcal{B}$ which we construct are as follows.

Adversary $\mathcal{B}(mpk : \text{PKGEN}, \text{DKGEN}, \text{LR})$

$Cr, Ch, SK \leftarrow \emptyset$

$fs, RT_{\text{rd}}, RT_{\text{wr}} \leftarrow \emptyset$; $ctr \leftarrow 1$

foreach $r \in R$:

$RT_{\text{rd}}[r] \leftarrow ctr$; $ctr \leftarrow ctr + 1$

foreach $r \in R$:

$RT_{\text{wr}}[r] \leftarrow ctr$; $ctr \leftarrow ctr + 1$

Query : $dk_A \leftarrow \text{DKGEN}(f_A)$

$State \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$; $fs[0][0] \leftarrow mpk$

$b' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda : \widetilde{\mathcal{O}}_{\mathsf{p2r}^*})$

return b'

In $\mathcal{B}$'s simulation of $\mathbf{Exp}^{\mathsf{p2r}^*}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma]}(\lambda)$, the auxiliary algorithms ReEnc, ReSign and RoleUpdate work as follows.

Algorithm $\mathsf{ReEnc}(fs, o^*)$

1 : **if** $o^* \notin O$ **then**
2 : **return** fs
3 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
4 : **if** $i > 0$ **then**
5 : $m \leftarrow \mathsf{Dec}(dk_A, fs[o^*][i].ctx)$
6 : $fs[o^*][i].ctx \leftarrow\!\!\$\ \mathsf{Enc}(fs[o^*][0].pk, m \,\|\, i)$
7 : $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], fs[o^*][i].ctx \,\|\, i)$
8 : $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i + 1)$
9 : **return** fs

Algorithm $\mathsf{ReSign}(fs, o^*)$

1 : **if** $o \notin O$ **then**
2 : **return** fs
3 : $i \leftarrow \mathsf{FindValidEntry}(fs, o^*)$
4 : **if** $i > 0$ **then**
5 : $m \leftarrow fs[o^*][i].ctx \,\|\, i$
6 : $fs[o^*][i].sig \leftarrow\!\!\$\ \mathsf{Sign}(SK[o^*], m)$
7 : $fs \leftarrow \mathsf{EraseRest}(fs, o^*, i + 1)$
8 : **return** fs

Algorithm $\mathsf{RoleUpdate}(fs, r^*)$

$\{rdk_u\}_{u \in Cr} \leftarrow \emptyset$
$RT_{\mathrm{rd}}[r^*] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
foreach $((o, \mathtt{read}), r^*) \in PA$:
  $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o, \mathtt{read}), r) \in PA\}$
  Query : $fs[o][0].pk \leftarrow \mathrm{LR}(y, y)$
  $fs \leftarrow_{\$} \mathsf{ReEnc}(fs, o)$
  $PK[ctr'] \leftarrow fs[o^*][0].pk$
  $ctr' \leftarrow ctr' + 1$
foreach $u \in Cr$:
  if $(u, r^*) \in UA$ **then**
    $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u, r) \in UA\}$
    Query : $rdk_u \leftarrow \mathrm{DKGen}(f_x)$
return $(fs, \{(rdk_u, \emptyset)\}_{u \in Cr})$

$\mathcal{B}$ answers $\mathcal{A}$'s oracle calls as specified in Figure 4.15 and 4.16.

Then we argue that the way $\mathcal{B}$ carries out the simulation will not lead to its oracle returning an error. Recall that, in $\mathcal{B}$'s game, a list Ch is used to record all the pairs of attribute sets which have been queried to the oracle LR so far. There is also another list F maintained in the game, which is used to record all the predicates submitted to DKGen for decryption keys so far. The oracles will not return an error if for all $f \in F$ and all $(I'_0, I'_1) \in Ch$, it holds that $f(I'_0) = f(I'_1)$.

During the simulation, $\mathcal{B}$ calls DKGen for generating decryption keys only with the universe of attributes and the decryption keys for corrupt users. Thus, the list F contains a predicate with the universe of attributes and also the predicates associated to attribute sets of corrupt users' roles. Before $\mathcal{A}$ calls the challenge oracle, $\mathcal{B}$ queries LR only for generating public keys with pairs of identical attribute sets. Therefore, the list Ch (in $\mathcal{B}$'s game) only contains pairs of same attribute sets at this stage. Thus, for all $f \in F, (I'_0, I'_1) \in Ch$, we have that $f(I'_0) = f(I'_1)$ is always satisfied before $\mathcal{A}$ calls ChllPA.

When $\mathcal{A}$ specifies its challenge (p, p, r_0, r_1) by calling ChllPA, $\mathcal{B}$ sends a pair of attribute sets (I_0, I_1) to LR, where I_b is the set of attributes related to the roles that have the permission p after the execution of $\mathsf{GrantPerm}$ with (p, r_b) individually. From the specification of LR, it is required that no corrupt user can have either the role r_0 or r_1. Thus the attributes associated to the two roles will not exist in any predicate in F. If there ever exists some corrupt user belonging to any of the two roles, the attribute will be renewed after the user is deassigned from the role. Moreover, $f_A(I_0) = f_A(I_1)$ is clearly satisfied since A is the universe of attributes.

Oracle AddUser(u^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $u \in U$ **then return** $\perp$
- $U \leftarrow U \cup \{u^*\}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AddObject(o^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $o \in O$ **then return** $\perp$
- $O \leftarrow O \cup \{o^*\}$
- $P \leftarrow P \cup \{(o^*, \mathtt{read}), (o^*, \mathtt{write})\}$
- $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
- **Query** : $fs[o^*][0].pk \leftarrow \mathrm{LR}(y, y)$
- $(sk_{o^*}, vk_{o^*}) \leftarrow_{\$} \mathsf{KeyGen}(1^\lambda)$
- $fs[o^*][0].vk \leftarrow vk_{o^*}; SK[o^*] \leftarrow sk_{o^*}$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle AssignUser(u^*, r^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $u^* \notin U \vee r^* \notin R \vee (u^*, r^*) \in UA$ **then return** $\perp$
- $UA \leftarrow UA \cup \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$} \mathsf{RoleUpdate}(fs, r^*)$
- $x \leftarrow \{RT_{\mathrm{wr}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $wdk \leftarrow \mathrm{DKGen}(f_x)$
- $msg_{u^*}.wdk \leftarrow wdk$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle DeassignUser(u^*, r^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $(u^*, r^*) \notin UA$ **then return** $\perp$
- $UA \leftarrow UA \setminus \{(u^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$} \mathsf{RoleUpdate}(fs, r^*)$
- **foreach** $((o, \mathtt{write}), r^*) \in PA$:
  - $(sk_o, vk_o) \leftarrow_{\$} \mathsf{KeyGen}(1^\lambda)$
  - $SK[o] \leftarrow sk_o; fs \leftarrow_{\$} \mathsf{ReSign}(fs, o)$
  - $fs[o][0].vk \leftarrow vk_o$
  - $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o, \mathtt{write}), r) \in PA\}$
  - **else** $fs[o][0].sk \leftarrow_{\$} \mathsf{Enc}(pk, sk_o)$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle GrantPerm(p^*, r^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $p^* \notin P \vee r^* \notin R \vee (p^*, r^*) \in PA$ **then return** $\perp$
- Parse p^* as $(o^*, mode)$
- $PA \leftarrow PA \cup \{(p^*, r^*)\}$
- **if** $mode = \mathtt{read}$ **then**
  - $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \mathtt{read}), r) \in PA\}$
  - **Query** : $fs[o^*][0].pk \leftarrow \mathrm{PKGen}(y, y)$
  - $fs \leftarrow_{\$} \mathsf{ReEnc}(fs, o^*)$
- **if** $mode = \mathtt{write}$ **then**
  - $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o^*, \mathtt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \mathrm{PKGen}(y, y)$
  - $fs[o^*][0].sk \leftarrow_{\$} \mathsf{Enc}(pk, SK[o^*])$
- **return** $(fs, \{\emptyset\}_{u \in Cr})$

Oracle RevokePerm(p^*, r^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $(p^*, r^*) \notin PA$ **then return** $\perp$
- $PA \leftarrow PA \setminus \{(p^*, r^*)\}$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- Parse p^* as $(o^*, mode)$
- **if** $mode = \mathtt{read}$ **then**
  - $y \leftarrow \{RT_{\mathrm{rd}}[r] \mid ((o^*, \mathtt{read}), r) \in PA\}$
  - **if** $y = \emptyset$ **then**
    - $y \leftarrow \{ctr\}; ctr \leftarrow ctr + 1$
  - **Query** : $fs[o^*][0].pk \leftarrow \mathrm{PKGen}(y, y)$
  - $fs \leftarrow_{\$} \mathsf{ReEnc}(fs, o^*)$
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$} \mathsf{RoleUpdate}(fs, r^*)$
- **if** $mode = \mathtt{write}$ **then**
  - $(sk_{o^*}, vk_{o^*}) \leftarrow_{\$} \mathsf{KeyGen}(1^\lambda)$
  - $SK[o^*] \leftarrow sk_{o^*}; fs \leftarrow_{\$} \mathsf{ReSign}(fs, o^*)$
  - $fs[o^*][0].vk \leftarrow vk_{o^*}$
  - $y \leftarrow \{RT_{\mathrm{wr}}[r] \mid ((o^*, \mathtt{write}), r) \in PA\}$
  - **Query** : $pk \leftarrow \mathrm{PKGen}(y, y)$
  - $fs[o^*][0].sk \leftarrow_{\$} \mathsf{Enc}(pk, sk_{o^*})$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Figure 4.15: $\widetilde{\mathcal{O}}_{\mathsf{p2r}^*}$ (part 1)

Thus we have, no matter whether the challenge permission specified by $\mathcal{A}$ is a write or a read permission, after the execution of GrantPerm with any of the two speicifed roles, $f(I_0) = f(I_1)$ still holds for all $f \in F$. Therefore, all $\mathcal{A}$'s queries will not lead to $\mathcal{B}$'s oracle return an error.

So we conclude that the simulation provided by $\mathcal{B}$ is perfect. In addition, the simu-

Oracle DELOBJECT(o^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $o^* \notin O$ **then return** $\perp$
- $\{msg_u\}_{u \in Cr} \leftarrow \emptyset$
- **foreach** $(p, r) \in PA$:
  - **if** $p \in \{(o^*, \mathtt{read}), (o^*, \mathtt{write})\}$ **then**
    - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$} \text{REVOKEPERM}(p, r)$
- $O \leftarrow O \setminus \{o^*\}$
- $P \leftarrow P \setminus \{(o^*, \mathtt{read}), (o^*, \mathtt{write})\}$
- $SK[o^*] \leftarrow \emptyset$
- $fs \leftarrow \mathsf{EraseRest}(fs, o^*, 0)$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

Oracle WRITE(u^*, o^*, m)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $\neg\mathsf{HasAccess}(u^*, (o, \mathtt{write}))$ **then**
  - **return** $\perp$
- $i \leftarrow \mathsf{GetLength}(fs, o^*)$
- $ctx \leftarrow_{\$} \mathsf{Enc}(fs[o^*][0].pk, m \,\|\, i+1 \,\|\, o^*)$
- $fs[o^*][i+1].ctx \leftarrow ctx$
- $sig \leftarrow_{\$} \mathsf{Sign}(SK[o^*], ctx \,\|\, i+1)$
- $fs[o^*][i+1].sig \leftarrow sig$
- **return** fs

Oracle CORRUPTU(u^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $u \notin U$ **then return** $\perp$
- $Cr \leftarrow Cr \cup \{u^*\}$
- $x \leftarrow \{RT_{\mathrm{rd}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $rdk \leftarrow \text{DKGEN}(f_x)$
- $x' \leftarrow \{RT_{\mathrm{wr}}[r] \mid (u^*, r) \in UA\}$
- **Query** : $wdk \leftarrow \text{DKGEN}(f_{x'})$
- **return** (rdk, wdk)

Oracle DELUSER(u^*)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $u^* \notin U$ **then return** $\perp$
- **foreach** $(u^*, r) \in UA$:
  - $(fs, \{msg_u\}_{u \in Cr}) \leftarrow_{\$} \mathsf{DeassignUser}(fs, u^*, r)$
- $U \leftarrow U \setminus \{u^*\}$; $Cr \leftarrow Cr \setminus \{u^*\}$
- **return** $(fs, \{msg_u\}_{u \in Cr})$

CHLLPA($Cmd, (p_0, p_1, r_0, r_1)$)
- **if** $\mathsf{challd} = 1$ **then return** $\perp$
- **if** $Cmd \neq \mathsf{GrantPerm} \vee p_0 \neq p_1$ **then**
  - **return** $\perp$
- **foreach** $u \in Cr$:
  - **if** $(u, r_0) \in UA \vee (u, r_1) \in UA$ **then**
    - **return** $\perp$
- Parse p_0 as $(o, mode)$
- **if** $mode = \mathtt{read}$ **then**
  - $y_0 \leftarrow \{RT_{\mathrm{rd}}[r] \mid (p_0, r) \in PA \vee r = r_0\}$
  - $y_1 \leftarrow \{RT_{\mathrm{rd}}[r] \mid (p_0, r) \in PA \vee r = r_1\}$
  - **Query** : $fs[o][0].pk \leftarrow \text{LR}(y_0, y_1)$
  - $fs \leftarrow \mathsf{ReEnc}(o)$
- **if** $mode = \mathtt{write}$ **then**
  - $y_0 \leftarrow \{RT_{\mathrm{wr}}[r] \mid (p_0, r) \in PA \vee r = r_0\}$
  - $y_1 \leftarrow \{RT_{\mathrm{wr}}[r] \mid (p_0, r) \in PA \vee r = r_1\}$
  - **Query** : $pk \leftarrow \text{LR}(y_0, y_1)$
  - $fs[o][0].sk \leftarrow_{\$} \mathsf{Enc}(pk, SK[o])$
- $\mathsf{challd} \leftarrow 1$
- **return** $(fs, \{msg_u\}_{u \in U})$

FS($query$)
- **if** $query =$ "STATE" **then**
  - **return** fs
- **if** $query =$ "APPEND($info$)" **then**
  - $fs \leftarrow fs \| info$; **return** fs

Figure 4.16: $\widetilde{\mathcal{O}}_{\mathsf{p2r}^*}$ (part 2)

lation is determined by the random bit chosen in $\mathcal{B}$'s own game. Thus, if $\mathcal{A}$ guesses the bit correctly, $\mathcal{B}$ guesses it correctly with the same probability, it holds:

$$\mathbf{Adv}^{id\text{-}h\text{-}pk}_{\mathcal{PE},\mathcal{B}}(\lambda) = \mathbf{Adv}^{\mathsf{p2r}^*-privacy}_{\mathcal{CRBAC}[\mathcal{PE},\Sigma],\mathcal{A}}(\lambda),$$

and therefore the theorem is proved. □

4.8 Conclusion

In this chapter, we presented the existing game-based security definitions for cRBAC systems. More specifically, the existing security definitions from previous work for correctness and secure read access were both redefine within the extended system model where access control on write access is supported. We also provide two formal security defintions for write access: the first one is with respect to secure access and the other one is closely related to correctness.

One main contribution in this chapter was the formal definition of past confidentiality. As observed from the study in UC framework, there exists a gap between the existing security definition for secure read access and the specification of the access control policy being enforced. Therefore, the purpose of defining past confidentiality serves as an atempt towards briding the gap.

The other main contribution is the rigorous security definition of policy privacy. When we use cryptographic techniques to enforce access control policies, the information about the policies might be unintentionally revealed. The security definition allows one to formally prove that such sensitive information will not be leaked during the system execution.

Finally, we provided a construction of the cRBAC system based on a novel type of predicate encryption scheme. The construction is proven to be secure under the definitions with respect to secure access and to preserve policy privacy to a certain degree.

Chapter 5

UC security of cRBAC

The content of this chapter is adapted from the paper *Universally Composable Cryptographic Role-Based Access Control* [51]. The work was done in conjunction with Bogdan Warinschi. I am responsable for providing most of the results which include the security definition, main theorems and their proofs. For consistency with the previous chapters, the results presented in this chapter have been slightly changed from those in the published paper.

5.1 Introduction

The security definitions for cRBAC systems presented in the previous chapter use the so-called *game-based* approach. The game formalises the interactions between the system and an attacker against the system and rigorously clarifies what a security breach is (e.g. as an event occurs during the execution or distinguishability between two executions). The most significant advantage of definitions defined via this approach is simplicity. It usually only considers the stand-alone scenarios where the execution environments of the system are not taken into account, since the game must specify the information that an adversary can obtain when attacking the system. Therefore, its security guarantee might not be preserved when the system is employed in those environments along with unforeseen security threats. Moreover, for complex systems like cRBAC, it is always difficult to exhaustively enumerate the different security properties we desire from the system. Sometimes we may not even sure whether the proposed security definitions appropriately capture the desired security requirements or not.

In this chapter, we consider a definitional alternative, called *simulation-based* approach, that does not suffer from the above shortcomings mentioned above. Under

this paradigm, security is defined by comparing a system with an idealised version and demands that the real execution of a system reveals at most as much information is revealed by an ideal version of the system. As a consequence, the real system inherits all of the security properties of the ideal one, so there is no need to enumerate security properties separately. One important class of simulation-based security considers executions determined by an arbitrary environment (tasked, e.g. to provide inputs to and obtain outputs from the system), so security in this sense is *composable* in the sense that it is preserved in any environment in which the system is employed [12, 40, 48]. Unfortunately, simulation security is often difficult to establish and imposes stringent restrictions on the implementations which rule out constructions with no obvious weakness or, at the very least, require inefficient realisations [14, 54]. So far the only attempt to provide a simulation-based security definition for access control systems is the work of Halevi et al. [38]. They proposed a security security definition for access control in a specific distributed file storage system rather than a general model.

5.1.1 Our results

Security definition. Our first result is a universally composable security definition of cRBAC systems. The expected security guarantees are captured by ideal functionality named $\mathcal{F}_{\text{CRBAC}}$, which simply behaves as a server-mediated access control on the files being protected. Only authorised access request from users will be granted. This essentially requires that a cRBAC implementation should enforce the expected semantics of RBAC system [60].

Relation with existing definitions. Next, we study the relation between the existent game-based security definitions and the level of security that our definition entails. It is generally believed that, for the same cryptographic task, simulation-based security is stronger than game-based security, even if only because the former is supposed to capture all of the security properties expected of a system. As expected, we show that our definition is strictly stronger than the two existing game-based security definitions: secure read access (introduced in [28]) and secure write access (introduced in [27]).

Lower-bound for UC-secure cRBAC. Our main result is a gap between simulation-security and game-based security. More precisely, we show that it is impossible for a cryptographic RBAC system to be UC-secure. In technical terms, we show that the so-called commitment problem [14] occurs in the context of access control. Roughly,

the problem is that the simulator required by the security definition needs to produce valid looking encryptions of the objects that are protected without actually knowing the actual content of these objects (e.g. files). The problem is that when the adversary gains access to such a file (e.g. by corrupting a user who has access to this file), the simulator needs to produce a decryption key that explains the ciphertext as an encryption of some particular content which the simulator did not know when the ciphertext was created.

The commitment problem is usually due to adaptive corruption of parties, in access control the problem can also be due to the transient access to files as parties gain and lose access to files depending on administrative commands. Apart from this, in access control adaptive party corruption is more significant since the party corruption can be motivated by change on the access control policy. Therefore, while the commitment problem can sometimes be waved away whenever adaptive corruption is not a concern, in cryptographic access control the problem seems inherent and posses severe limitations if one aims for simulation-based security.

To summarise, in this chapter we make the following contributions:

- We propose a formal security definition for cRBAC in the UC framework.
- We study the relation between the UC definition and two existing definitions presented in Chapter 4.
- We show a lower bound for UC-secure cRBAC systems with adaptive corruption.

5.2 A UC Security Definition for cRBAC

In this section, we present a universally composable security definition for cRBAC systems. We formalise the security requirements by designing an ideal functionality $\mathcal{F}_{\text{CRBAC}}$.

5.2.1 Functionality $\mathcal{F}_{\text{CRBAC}}$

The ideal functionality we present in Figure 5.1 captures the intuitive security properties of cRBAC systems in the way of simply behaving as a server-mediated access control on the files being protected. Very roughly, $\mathcal{F}_{\text{CRBAC}}$ keeps track of every operation performed on the system and maintains the induced access control matrix within, while it preserves that only the authorised access requests will be granted. This is achieved by having $\mathcal{F}_{\text{CRBAC}}$ maintain a built-in database to store the content of every file, along with a symbolic RBAC state of the system. Then it handles every access request according to

Functionality $\mathcal{F}_{\text{CRBAC}}$

$\mathcal{F}_{\text{CRBAC}}$ proceeds as follow, with a manager M, users u_1, ..., u_n and an adversary $\mathcal{S}$.

Initialisation: Upon receiving an input $(\texttt{Initialisation}, sid, R)$ from M where R is a set of roles, send $(\texttt{Initialisation}, sid, R)$ to $\mathcal{S}$, initialise an object-indexed list $FS \leftarrow \emptyset$ and the symbolic RBAC state $(U, O, P, PA, UA) \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$. After that, mark the system as initialised and ignore all the inputs of the form $(\texttt{Initialisation}, sid, R')$ for any R' from now on.

RBAC administration: Upon receiving an input $(\texttt{RBAC}, sid, Cmd, arg)$ from M where Cmd is one of the administrative RBAC commands specified in Figure 2.2 and arg is the command-specific arguments, proceed as follows: if the system has not yet been initialised, or $\neg\mathsf{IsValid}(Cmd, arg)$ holds, return an error. Otherwise, execute the RBAC command symbolically by $(U, O, P, UA, PA) \leftarrow Cmd((U, O, P, UA, PA), arg)$. If $Cmd = \mathsf{DelObject}$ and $arg = o$, also delete the content stored in $FS[o]$. Then send $(\texttt{RBAC}, sid, Cmd, arg)$ to $\mathcal{S}$.

Write: Upon receiving an input $(\texttt{Write}, sid, o, m)$ from some party P where o is an object and m is some content, returns an error if the system has not been initialised. If P is some user u such that $\mathsf{HasAccess}(u, (o, \texttt{write}))$ is not satisfied, returns an error; otherwise, set $FS[o] \leftarrow m$. If there exists a corrupt user $u' \in U$ such that $\mathsf{HasAccess}(u', (o, \texttt{read}))$, send a message $(\texttt{Wrote}, sid, o, m)$ to $\mathcal{S}$; otherwise, send $(\texttt{Wrote}, sid, o, |m|)$ instead, where $|m|$ is the length of m.

Read: Upon receiving an input $(\texttt{Read}, sid, o)$ from some user u where o is an object, if the system has not been initialised or $\mathsf{HasAccess}(u, (o, \texttt{read}))$ is not satisfied, return an error; otherwise, set $m \leftarrow FS[o]$ (if $FS[o]$ stores no content then set m as an empty value). If there exists a corrupt user $u' \in U$ such that $\mathsf{HasAccess}(u', (o, \texttt{write}))$, send a message $(\texttt{Choose-value}, sid, u, o)$ to the adversary. Upon receiving a message $(\texttt{Value}, sid, m')$ from $\mathcal{S}$, set $m = m'$; upon receiving a message $(\texttt{Proceed}, sid)$ from $\mathcal{S}$, it does not change the value of m. Then return m to u.

Corruption: $\mathcal{F}_{\text{CRBAC}}$ is a standard corruption ideal functionality, with the exception that any request for corrupting M will be ignored.

Figure 5.1: Ideal functionality for cryptographic Role-Based Access Control, $\mathcal{F}_{\text{CRBAC}}$.

the RBAC state.

More specifically, $\mathcal{F}_{\text{CRBAC}}$ embodies the essential interfaces of a cRBAC system, including system initialisation, RBAC administration and read/write access to the file system. It proceeds as follows. Having received an initialisation request with a set of roles R from the manager M, $\mathcal{F}_{\text{CRBAC}}$ initialises an object-indexed list FS and the symbolic system RBAC state. Then it notices the adversary that the access control system is initialised with the set of roles R. Once $\mathcal{F}_{\text{CRBAC}}$ is initialised, it ignores the other

initialisation request afterwards. Having received a request of executing an administrative RBAC command from M, $\mathcal{F}_{\text{CRBAC}}$ checks if the execution of command and its arguments specified in the request is valid. If so, it executes the command symbolically and updates the maintained system RBAC state. The administrative RBAC command can be either of the commands presented in Section 2.7. Having received a request to write some content m on a file o from some party P, if P is a user and it has the write permission of o or P is the manager, $\mathcal{F}_{\text{CRBAC}}$ stores m in $FS[o]$ and leaks the file name o and the length of m to the adversary. Otherwise, it returns an error. Having received a request to read the content of a file o from some party P, if P is a user and it has the read permission of o or P is the manager, $\mathcal{F}_{\text{CRBAC}}$ sets m as the content stored in $FS[o]$. If $FS[o]$ stores no content, m is set to be the empty string ϵ. In the case that there exists a corrupt user who has the write permission of o, $\mathcal{F}_{\text{CRBAC}}$ asks the adversary for providing the file content that u can read. If $\mathcal{S}$ provides a value m', $\mathcal{F}_{\text{CRBAC}}$ replaces the value of m by m'. Then $\mathcal{F}_{\text{CRBAC}}$ returns m to u. $\mathcal{F}_{\text{CRBAC}}$ is a standard corruption ideal functionality, with an exception that the manager M cannot be corrupted. It captures the reasonable trust on the manager to administrate the access control system.

Several remarks on $\mathcal{F}_{\text{CRBAC}}$ are in order. First, $\mathcal{F}_{\text{CRBAC}}$ is an ideal functionality for cryptographic enforcement of (core) role-based access controls. Due to the purpose of studying the relation between the existing game-based security notions, $\mathcal{F}_{\text{CRBAC}}$ does not handle any administrative command of adding a new role to the system or removing an existing role from it. Second, $\mathcal{F}_{\text{CRBAC}}$ only guarantees secure access to the file system and preserves no policy privacy (when handling an administrative request, it simply reveals the command and its arguments to the adversary). There are still some design choices available on policy privacy preserving (e.g. only leaking the executed command but not its arguments to the adversary), which is left as further study. Third, $\mathcal{F}_{\text{CRBAC}}$ makes no explicit restriction on the form of the file system. Moreover, the file system is not designed as an individual party of the system. Thus in a real-world cRBAC system, the file system should be implemented by the protocol itself. It also captures that the file system does not implement any access control mechanism. Fourth, $\mathcal{F}_{\text{CRBAC}}$ does not have any authentication mechanism on the parties' identities. The authentication is left to the protocols that make calls to $\mathcal{F}_{\text{CRBAC}}$.

5.2.2 The Associated Protocol

Before presenting our definition of universally composable cRBAC system, we first need to transform a cRBAC scheme $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm) into an associated protocol $\Pi_{\mathcal{CRBAC}}$ in the UC setting. Recall that in a cRBAC system, private channels are assumed between the manager and the users. To model this, we let the parties have access to $\mathcal{F}_{\text{SMT}}$, the ideal functionality of secure message transmission which is presented in Figure 2.1. Also, $\mathcal{CRBAC}$ makes use of a public-accessible versioning file system. This is modelled by an appropriate functionality $\mathcal{F}_{\text{VFS}}$ which is presented in Figure 5.2.

The ideal functionality $\mathcal{F}_{\text{VFS}}$ proceeds with a set of users and a data manager. Essentially, it serves as an ideal versioning file system which guarantees the correct ordering of the file versions. The users can "write" to the file system by appending new versions to the files instead of overwriting existing contents. The data manager is provided with richer interfaces: it can remove and even rewrite some existing version of a file. All the users in the system can check the current state of the file system by providing a status request to $\mathcal{F}_{\text{VFS}}$. In implementation, the state of the file system would be a bitstring which consists of an array of (possibly encrypted) files; while in $\mathcal{F}_{\text{VFS}}$, it is presented as a list of entries with no loss of generality. When any change happens to the file system, the ideal functionality reveals the change to the adversary and also notices the users about the change. These reflect the public-accessible feature of the file system. In addition, any write operation to the file system is done in an anonymous manner, $\mathcal{F}_{\text{VFS}}$ will not reveal information about the identity of the party who carries out the write operation.

To simplify the protocol presentation, we define the following shorthand notations. When a party runs an cRBAC algorithm, it may generate a set of order-preserving instructions to be carried out on the file system. We use $\{info_i\}_{i\in\mathbb{N}}$ to denote such a set of instructions. If the party is the manager, each instruction $info_i \in \{info_i\}_{i\in\mathbb{N}}$ can be either $(\texttt{Write}, sid, o, ver, c)$ or $(\texttt{Remove}, sid, o, ver)$, where sid is the session id of $\mathcal{F}_{\text{VFS}}$. If the party is a user, it can only be the form $(\texttt{Write}, sid, o, c)$. A party may also need to come up with a set of order-preserving instructions $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$ such that after carrying out the instructions on the file system in order, the current state of the file system fs would become fs'. We say a party sends $\{info_i\}_{i\in\mathbb{N}}$ (or $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$) to $\mathcal{F}_{\text{VFS}}$, it means the party provides every instruction $info_i$ of the set as the input to $\mathcal{F}_{\text{VFS}}$ in order.

We now present the associated protocol $\Pi_{\mathcal{CRBAC}}$ (in Figure 5.3) and define universally

Functionality $\mathcal{F}_{\text{VFS}}$

$\mathcal{F}_{\text{VFS}}$ proceeds as follows, running with users $u_1, ..., u_n$, a file system manager M and an adversary $\mathcal{S}$. At the first activation $\mathcal{F}_{\text{VFS}}$ initialises a list L to be empty.

Status: Upon receiving an input $(\texttt{Status}, sid)$ from a party P, output $(\texttt{Content}, sid, L)$ to P.

Write (user): Upon receiving an input $(\texttt{Write}, sid, o, c)$ from some user u, if no record $r \in L$ of the form $(sid, o, \cdot, \cdot)$ exists, set $L \leftarrow L \cup \{(sid, o, 1, c)\}$ and set $ver = 1$; otherwise, set $L \leftarrow L \cup \{(sid, o, ver_m + 1, c)\}$ and set $ver = ver_m + 1$, where $ver_m = max(\{ver | (sid, o, ver, \cdot) \in L\})$. Then send $(\texttt{Wrote}, sid, o, ver, c)$ to $\mathcal{S}$, and send $(\texttt{Updated}, sid)$ to every user.

Write (manager): Upon receiving an input $(\texttt{Write}, sid, o, ver, c)$ from M, if a record $r \in L$ of the form $(sid, o, ver, \cdot)$ exists, modify r as (sid, o, ver, c); otherwise, set $L \leftarrow L \cup \{(sid, o, ver, c)\}$. Then send $(\texttt{Wrote}, sid, o, ver, c)$ to $\mathcal{S}$ and send $(\texttt{Updated}, sid)$ to every user.

Remove: Upon receiving an input $(\texttt{Remove}, sid, o, ver)$ from M, set $L \leftarrow L \setminus \{(sid, o, ver, c)\}$. Then send $(\texttt{Removed}, sid, o, ver, c)$ to $\mathcal{S}$ and send $(\texttt{Updated}, sid)$ to every user.

Figure 5.2: Ideal functionality for versioning file storage, $\mathcal{F}_{\text{VFS}}$.

composable cRBAC system.

Now we can define UC security of cRBAC systems.

Definition 13. *A cRBAC system defined by the scheme $\mathcal{CRBAC}$ =* (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Read, Write, Update) *is UC-secure if the associated protocol $\Pi_{\mathcal{CRBAC}}$ securely realises $\mathcal{F}_{\text{CRBAC}}$ in $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model and in the setting that the manager never gets corrupted.*

5.3 UC security is stronger than Game-Based Security

Based on the transformation above, we now study the relation between UC security and two existing game-based security definitions: secure read access (Definition 8) and secure write access (Definition 10). We treat security of read access separately from that of write access.

Theorem 7. *Any cRBAC system defined by the scheme $\mathcal{CRBAC}$ =* (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm) *which is UC-secure (in $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model) is secure with respect to write access.*

Proof. We prove this theorem by showing that if a cRBAC system which is not secure with respect to write access, then it cannot be UC-secure. Assumes that a cRBAC sys-

The Protocol $\Pi_{\mathcal{CRBAC}}$

The participants: a manger M and a set of users $u_1, ..., u_n$.

Initialisation: Upon receiving an input $(\texttt{Initialisation}, sid, R)$ where R is a set of roles, M computes $(st_M, fs, \{msg_u\}_{u\in U}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$. It then invokes an instance of $\mathcal{F}_{\text{VFS}}$ as the data manager with session id (M, sid), parses fs as $\{info_i\}_{i\in\mathbb{N}}$ and sends $\{info_i\}_{i\in\mathbb{N}}$ to $\mathcal{F}_{\text{VFS}}$. If $\{msg_u\}_{u\in U}$ is non-empty, M sends msg_u to every user u using $\mathcal{F}_{\text{SMT}}$.

Administration: Upon receiving an input $(\texttt{RBAC}, sid, Cmd, arg)$ where Cmd can be either of the administrative commands specified in Session 2.7 and arg is the arguments of the command. If $\mathsf{IsValid}(Cmd, arg)$ holds, M sends $(\texttt{Status}, (M, sid))$ to $\mathcal{F}_{\text{VFS}}$ to obtain $(\texttt{Content}, sid, fs)$ and then computes $(st'_M, fs', \{msg_u\}_{u\in U}) \leftarrow\!\!\$\ cmd(st_M, fs, arg)$, where cmd is the algorithm that implements the administrative command Cmd. M sets $st_M \leftarrow st'_M$, and then comes up with $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$. If $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$ is non-empty, M sends $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$ to $\mathcal{F}_{\text{VFS}}$. If $\{msg_u\}_{u\in U}$ is non-empty, M sends msg_u to every user u using $\mathcal{F}_{\text{SMT}}$.

Update: Upon receiving a message $(\texttt{Update}, sid, msg_u)$ from M, a user u computes $st'_u \leftarrow\!\!\$\ \mathsf{Update}(st_u, msg_u)$, where st_u is u's local state (st_u is an empty value when u receives the first update message from M). Then it sets $st_u \leftarrow st'_u$.

Write: Upon receiving an input $(\texttt{Write}, sid, o, m)$, a user u sends $(\texttt{Status}, (M, sid))$ to $\mathcal{F}_{\text{VFS}}$ to get $(\texttt{Content}, sid, fs)$ and computes $fs' \leftarrow\!\!\$\ \mathsf{Write}(st_u, fs, o, m)$. Then u comes up with $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$ and sends it to $\mathcal{F}_{\text{VFS}}$.

Read: Upon receiving an input $(\texttt{Read}, sid, o)$, a user u sends $(\texttt{Status}, (M, sid))$ to $\mathcal{F}_{\text{VFS}}$ to get $(\texttt{Content}, sid, fs)$ and then outputs $(\texttt{Read}, sid, \mathsf{Read}(st_u, fs, o))$.

Figure 5.3: The Protocol $\Pi_{\mathcal{CRBAC}}$ in $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model.

tem defined by the scheme $\mathcal{CRBAC}$ is not secure with respect to write access, then there exists an adversary $\mathcal{A}_W$ which can win in the game that defines secure write access with non-negligible probability. We show that given such an adversary, an environment $\mathcal{Z}$ can be constructed to distinguish its interactions with the associated protocol $\Pi_{\mathcal{CRBAC}}$ and a dummy adversary $\mathcal{D}$, from the interactions with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ and a simulator $\mathcal{S}$ with non-negligible probability. Due to the subroutine respecting requirement, for simplicity, we make a mild assumption that in the experiment $\mathbf{Exp}^{\text{write}}_{\mathcal{CRBAC},\mathcal{A}_W}$, the adversary can append contents to a file by calling FS only when there exists some corrupt user in the system. It restricts that the file system is publicly acceesible only to the users in the system.

We now describe how $\mathcal{Z}$ works. During its execution, $\mathcal{Z}$ maintains three lists: an object-indexed list T for recording the last valid file contents written by users, fs for recording the current state of the file system and Cr for recording the corrupt users.

$\mathcal{Z}$ first activates the manager M with an input $(\texttt{Initialization}, sid, R)$, where sid is an arbitrary string and R is a random set of roles, to initialise the cRBAC system. It then obtains a sequence of messages regarding the changes on $\mathcal{F}_{\text{VFS}}$ (via the dummy adversary $\mathcal{D}$ who just simply delivers the messages) and updates fs accordingly so that fs is identical to the list L maintained by $\mathcal{F}_{\text{VFS}}$. Then $\mathcal{Z}$ runs a local copy of $\mathcal{A}_W$ and starts to simulate $\mathbf{Exp}^{\text{write}}_{\mathcal{CRBAC},\mathcal{A}_W}$ as follows.

1. When $\mathcal{A}_W$ asks for executing any RBAC command Cmd with arguments arg, if the execution of Cmd with arg is valid, $\mathcal{Z}$ activates M with an input $(\texttt{RBAC}, sid, cmd, arg)$ and updates fs according to the messages received from $\mathcal{F}_{\text{VFS}}$ (via $\mathcal{D}$). If the execution of the command will lead to any user in the list Cr has the write permission of any file o, $\mathcal{Z}$ sets $T[o]$ as a special value $\texttt{adv}$. If M sends update messages to the corrupt users when executing the command, $\mathcal{Z}$ will receive the update messages (via $\mathcal{D}$) and then forward them to $\mathcal{A}_W$. In addition, if $\mathcal{A}_W$ requests to delete some user which is in the list Cr, $\mathcal{Z}$ removes this user from Cr after executing the command. If $\mathcal{A}_W$ requests to delete some file o, the content in $T[o]$ will be also deleted. Finally, $\mathcal{Z}$ hands fs to $\mathcal{A}_W$.

2. When $\mathcal{A}_W$ requests an honest user u to write some content m to a file o, if u does not have the write permission of o, $\mathcal{Z}$ returns an error; otherwise $\mathcal{Z}$ activates u with an input $(\texttt{Write}, sid, o, m)$ and updates fs according to the messages received from $\mathcal{F}_{\text{VFS}}$ (via $\mathcal{D}$). Then $\mathcal{Z}$ hands fs to $\mathcal{A}_W$. If there exists no user in the list Cr that has write access to o, $\mathcal{Z}$ sets $T[o]$ as m.

3. When $\mathcal{A}_W$ requests for corrupting a user u, $\mathcal{Z}$ corrupts u (via $\mathcal{D}$) and forwards the obtained local state to $\mathcal{A}_W$. For every file o to which u has write access, $\mathcal{Z}$ sets $T[o]$ as the special value $\texttt{adv}$. Then it adds u to Cr.

4. When $\mathcal{A}_W$ queries the current state of the file system, $\mathcal{Z}$ hands fs to $\mathcal{A}_W$.

5. When $\mathcal{A}_W$ requests to update the file system with some information $info$, $\mathcal{Z}$ parses $info$ as (o, c), where o is a file name and c is the content to be appended to the file system. $\mathcal{Z}$ chooses a corrupt user u and sends u a message (via $\mathcal{D}$) to let it provide an input $(\texttt{Write}, sid', o, c)$ to $\mathcal{F}_{\text{VFS}}$, where sid' is the session id of $\mathcal{F}_{\text{VFS}}$. Then $\mathcal{Z}$ updates the fs according to the messages received from $\mathcal{F}_{\text{VFS}}$ and hands it to $\mathcal{A}_W$. If such a corrupt user u does not exist, $\mathcal{Z}$ just returns an error.

6. When $\mathcal{A}_W$ outputs a target file o^*, $\mathcal{Z}$ activates M with $(\texttt{Read}, sid, o^*)$ to obtain

the output m. If both $T[o^*] \neq \texttt{adv}$ and $T[o^*] \neq m$ are satisfied, $\mathcal{Z}$ outputs 1; otherwise it outputs 0.

We now discuss $\mathcal{Z}$'s outputs in the two worlds separately. In the case that $\mathcal{Z}$ interacts with real-world execution of $\Pi_{\mathcal{CRBAC}}$ and $\mathcal{D}$, from $\mathcal{A}_W$'s perspective, $\mathcal{Z}$'s simulation is indistinguishable from the real experiment. Therefore, by assumption $\mathcal{A}_W$ should have written some valid content to the file system without having the permission with non-negligible probability. Hence $\mathcal{Z}$ will output 1 with the same probability.

If $\mathcal{Z}$ interacts with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ and $\mathcal{S}$, we show that $\mathcal{Z}$ will always output 0 since $\mathcal{A}_W$ can never win in this case. First recall that, in order to win the write security game, when $\mathcal{A}_W$ terminates with an output o^* the following two conditions must hold: (1) $T[o^*]$ must not equal to the special value `adv` (the experiment maintains an invariant that if there exists any corrupt user has the write permission of some file o, the list $T[o]$ must be the special value `adv`) and (2) the current content of o^* (read by M) must be different from the record in $T[o^*]$. Next we discuss that the above two winning conditions cannot be both satisfied when $\mathcal{A}_W$ generates its output.

Suppose that condition (1) holds when $\mathcal{A}_W$ outputs o^*. Since $T[o^*] \neq \texttt{adv}$, $T[o^*]$ can be one of the two possible values, either an empty value ϵ or the content written by the last operation to o by some honest user who has the write permission (otherwise $\mathcal{Z}$ will not record that in $T[o^*]$). In the first case, $T[o^*]$ is an empty value implies that $\mathcal{Z}$ has not yet handled any write request to o^* since the recent initialisation of o^* (o^* might have been deleted before but it is added back to the system later). Therefore, the value of $FS[o^*]$ in $\mathcal{F}_{\text{CRBAC}}$ would be also an empty value. From the specification of $\mathcal{F}_{\text{CRBAC}}$, it is clear that when $\mathcal{Z}$ activates M with the input $(\texttt{Read}, sid, o^*)$, it will obtain the content stored in $FS[o^*]$ which is the empty value here. Thus we have, the content read by u must equal to the record in $T[o^*]$ in this case. For the other possibility, if $T[o^*]$ equals to the content m which is written by the last write operation to o by some honest user, $\mathcal{Z}$ should have activated that user with an input $(\texttt{Write}, sid, o^*, m)$ when $\mathcal{A}_W$ requests for this write operation. Once $\mathcal{F}_{\text{CRBAC}}$ receives such an input, it stored m in $FS[o^*]$. Then when $\mathcal{Z}$ activates the manager with an input $(\texttt{Read}, sid, o^*)$, $\mathcal{F}_{\text{CRBAC}}$ will always return m in this case, Thus the content read by M also equals to the record in $T[o^*]$.

So, if $T[o^*] \neq \texttt{adv}$, $T[o^*]$ must equal to the content read by the manager, which means the two winning conditions can never be both satisfied. Thus, if $\mathcal{Z}$ interacts with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ and $\mathcal{S}$, $\mathcal{A}_W$ can never win in the simulated experiment and

$\mathcal{Z}$ will output 1 with probability 0.

Finally, it can be concluded that $\mathcal{Z}$'s outputs in the two worlds differ by a non-negligible amount, which means $\Pi_{\mathcal{CRBAC}}$ does not securely realise $\mathcal{F}_{\text{CRBAC}}$ and the theorem is proved.

Theorem 8. *Any cRBAC system defined by the scheme* $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm) *which is UC-secure (in* $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$*-hybrid model) is secure with respect to read access.*

Proof idea. The proof idea of this theorem is analogous to Theorem 7's. Given an adversary $\mathcal{A}_R$ that breaks read security of the cRBAC system defined by the scheme $\mathcal{CRBAC}$ with non-negligible probability, an environment $\mathcal{Z}$ can be constructed to tell its interactions with the execution of $\Pi_{\mathcal{CRBAC}}$ and a dummy adversary from the interactions with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ and a simulator. Similarly, $\mathcal{Z}$ runs a local copy of $\mathcal{A}_R$ and simulates to it the experiment $\mathbf{Exp}^{\text{read}}_{\mathcal{CRBAC},\mathcal{A}_R}$. $\mathcal{Z}$ first selects a random bit $b \leftarrow\!\!\$\ \{0,1\}$. When $\mathcal{A}_R$ specifies his challenge, $\mathcal{Z}$ writes to the file according to the value of b. Then $\mathcal{Z}$ transforms every query from $\mathcal{A}_R$, which will not lead to any corrupt user can get read access to any challenge file, into appropriate inputs being provided to the parties and the adversary.

In the case that $\mathcal{Z}$ is interacting with the real-world execution of $\Pi_{\mathcal{CRBAC}}$ and $\mathcal{D}$, from $\mathcal{A}_R$'s perspective, $\mathcal{Z}$'s simulation would be identical to the real experiment. Therefore, by assumption on $\mathcal{A}_R$, $\mathcal{Z}$ outputs 1 with probability significantly greater than $\frac{1}{2}$. If $\mathcal{Z}$ is interacting with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ and $\mathcal{S}$, from the specification of $\mathcal{F}_{\text{CRBAC}}$, we can infer that the only way that $\mathcal{A}_R$ can learn some partial information about the contents of the files is to retrieve them via the authorised users. However, $\mathcal{Z}$ prevents all the corrupt users from being granted the read permission of any challenge file. Therefore $\mathcal{A}_R$ will not be able to learn any partial content of the challenge files which means the best it can do is to output a random guess. Thus, $\mathcal{Z}$ outputs 1 with probability $\frac{1}{2}$ in this case.

Finally, it can be concluded that $\mathcal{Z}$ would be able to distinguish its interactions in the two worlds with non-negligible probability, which means $\Pi_{\mathcal{CRBAC}}$ does not securely realise $\mathcal{F}_{\text{CRBAC}}$ in $\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}}$-hybrid model and therefore is not UC-secure. The proof of this theorem is also under the assumption that in the experiment $\mathbf{Exp}^{\text{read}}_{\mathcal{CRBAC},\mathcal{A}_R}$, the adversary can append contents to a file on its own only when there exists some corrupt user in the system.

Functionality $\mathcal{F}_{\text{NCE}}$

$\mathcal{F}_{\text{NCE}}$ works as follows. It interacts with a message sender P_S, a receiver P_R and an adversary $\mathcal{S}$.

Pre-processing phase: Upon receiving an input $(\texttt{Init}, sid, P_R)$ from P_S, send $(\texttt{Init}, sid, P_S)$ to P_R and send $(\texttt{Init}, sid, P_S, P_R)$ to $\mathcal{S}$. After that, mark the channel as established.

Communication phase: Upon receiving an input $(\texttt{Send}, sid, P_R, m)$ from P_S, if the channel has not been established, ignore this input. Otherwise, deliver the message $(\texttt{Send}, sid, P_S, m)$ to P_R and reveal $(\texttt{Sent}, sid, P_S, P_R, |m|)$ to $\mathcal{S}$, where $|m|$ is the length of the message.

Corruption: Upon receiving $(\texttt{Corrupt}, sid, P)$ from $\mathcal{S}$ where $P \in \{P_S, P_R\}$, reveal m to $\mathcal{S}$. If $P = P_S$ and the message has not yet been delivered to P_R, ask $\mathcal{S}$ for a value m' then output $(\texttt{Sent}, sid, P_S, m')$ to P_R.

Figure 5.4: Ideal functionality for non-committing encryption, $\mathcal{F}_{\text{NCE}}$ (adapted from [69]).

5.4 Impossibility of UC-secure cRBAC

In this section we present our main result. We show that the level of security demanded by a UC-secure cRBAC system cannot be achieved, even in a setting where the cRBAC system has access to an idealised file system and secure channels between all parties are assumed. Our impossibility result is in a setting where the adversary can adaptively corrupt honest protocol participants.

Theorem 9. *In the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model, there exists no UC-secure cRBAC system with adaptive corruptions.*

Proof: Our proof consists of two steps. First, we show that the existence of any UC-secure cRBAC system implies the existence of a universally composable non-interactive communication protocol (NICP). Specifically, we provide a generic construction of a NICP that securely realises the functionality $\mathcal{F}_{\text{NCE}}$ of non-committing encryption (which is presented in Figure 5.4), from any UC-secure cRBAC system. Next, we argue that the resulting communication protocol in fact cannot securely realise $\mathcal{F}_{\text{NCE}}$ – this step is an extension of a well-known result by Nielsen, to a setting where parties have access to a secure file system and secure channels. Thus, it contradicts the existence of the UC-secure cRBAC systems.

We start by describing the generic construction for the universally composable NICP. Recall that based on our transformation, the associated protocol of a cRBAC scheme works in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model and in a setting that the manager never gets

The Protocol Π_{NICP}

The participants: a message sender P_S, a receiver P_R and a trusted party M namely the manager.

Pre-processing phase. M establishes the communication channel for P_S and P_R. In this stage, some content might be written to $\mathcal{F}_{\text{VFS}}$ for the channel set-up.

1. Upon receiving an input $(\texttt{Init}, sid, P_R)$, P_S sends $(\texttt{Init}, sid, P_R)$ to M using $\mathcal{F}_{\text{SMT}}$.

2. Upon receiving a message $(\texttt{Init}, sid, P_R)$ from P_S, M selects a random role r and computes $(st_M, fs, \{st_{u_S}, st_{u_R}\}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, \{r\})$, where u_S and u_R are two users to be added to the system. It initialises two lists $msg_S \leftarrow st_{u_S}$ and $msg_R \leftarrow st_{u_R}$. M then invokes an instance of $\mathcal{F}_{\text{VFS}}$ with session id (M, sid) as the data manager and parses fs as $\{info_i\}_{i\in\mathbb{N}}$. If $\{info_i\}_{i\in\mathbb{N}}$ is non-empty, M sends $\{info_i\}_{i\in\mathbb{N}}$ to $\mathcal{F}_{\text{VFS}}$. After that, M runs a sequence of algorithms which implement the related administrative RBAC commands to add two users u_S, u_R and an object o to the system, to grant the write permission of o to u_S via the role r and to grant the read permission of o to u_R via r. The run of any of the algorithms might lead to the file system's current state fs gets updated to fs'. If so, M comes up with $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$ and sends it to $\mathcal{F}_{\text{VFS}}$. Whenever an update message msg for u_S (u_R resp.) is generated, M appends it to the list msg_S (msg_S resp.). Finally, after the run of the algorithms M sends $(\texttt{Update}, sid, msg_S)$ to P_S and sends $(\texttt{Update}, sid, msg_R)$ to P_R using $\mathcal{F}_{\text{SMT}}$.

3. Upon receiving a message $(\texttt{Update}, sid, msg_X)$ from M where $X \in \{S, R\}$, the party P_X updates its local state by running the update algorithm $st_X \leftarrow\!\!\$\ \mathsf{Update}(st_X, msg)$ on each update message $msg \in msg_X$ in order.

Communication Phase. Once the channel has been established, P_S can send arbitrarily many messages to P_R via $\mathcal{F}_{\text{VFS}}$.

1. Upon receiving an input $(\texttt{Send}, sid, P_R, m)$, P_S sends $(\texttt{Status}, (M, sid))$ to $\mathcal{F}_{\text{VFS}}$ to get $(\texttt{Content}, (M, sid), fs)$, and then computes $fs' \leftarrow\!\!\$\ \mathsf{Write}(st_S, fs, o, m)$. Next, P_S comes up with $\{info_i^{fs\to fs'}\}_{i\in\mathbb{N}}$ and sends it to $\mathcal{F}_{\text{VFS}}$.

2. Upon receiving an subroutine output $(\texttt{Updated}, (M, sid))$ from $\mathcal{F}_{\text{VFS}}$, P_R sends $(\texttt{Status}, (M, sid))$ to $\mathcal{F}_{\text{VFS}}$ to get $(\texttt{Content}, (M, sid), fs)$, and then outputs $m' \leftarrow \mathsf{Read}(st_R, fs, o)$.

Figure 5.5: The Protocol Π_{NICP} in $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model.

corrupted, the resulting communication protocol therefore works in the same hybrid model and makes use of such a trusted party in a restricted way.

The communication protocol involves a message sender, a receiver and a trusted party namely the manager. We demand that there exists no direct communication channel between the sender and the receiver. They have to communicate with each other in an indirect way: after a pre-processing phase in which the manager interacts with the other two parties over secure channels to establish the communication, the

sender can send messages to the receiver by writing to the file system and then the receiver performs read operations to get the messages. Notice that the read operation will not make any change to the file system, and the manager only works in the pre-processing phase and does not involve in the communication phase. The communication protocol in fact requires no interaction between the sender and the receiver. Hence it can be considered as non-interactive.

More specifically, let $\mathcal{CRBAC}$ = (Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Update, Write, Read) be the cRBAC scheme that defines the UC-secure cRBAC system. We denote the NICP by Π_{NICP} and present it Figure 5.5.

Then we show that Π_{NICP} securely realises $\mathcal{F}_{\text{NCE}}$ in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model. By assumption, the system cRBAC is UC-secure implies that there exists a simulator $\mathcal{S}$ such that no environment can tell with non-negligible probability whether it interacts with the parties running $\Pi_{\mathcal{CRBAC}}$ in the $(\mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{VFS}})$-hybrid model and a dummy adversary $\mathcal{D}$, or it interacts with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ with $\mathcal{S}$. Then we give the construction of the simulator $\mathcal{S}_{\text{NCE}}$ for Π_{NICP} as follows. $\mathcal{S}_{\text{NCE}}$ internally runs an instance of $\mathcal{S}$. Then it interacts with $\mathcal{S}$ as the environment and simulates to $\mathcal{S}$ the ideal process for $\mathcal{F}_{\text{CRBAC}}$. It proceeds as follow.

1. **Simulating the pre-processing phase.** Upon receiving from $\mathcal{F}_{\text{NCE}}$ a message $(\texttt{Init}, sid, P_S, P_R)$, $\mathcal{S}_{\text{NCE}}$ selects a random role r. It then simulates the pre-processing phase by sending messages to $\mathcal{S}$ sequentially in the name of $\mathcal{F}_{\text{CRBAC}}$ indicating that the cRBAC system is initialised with a role r, two users u_S and u_R, an object o are added to the system, u_S is granted the write permission of o via the role r and u_R is granted the read permission of o via r. When the environment requests $\mathcal{S}_{\text{NCE}}$ to provide any information that it can obtain during this phase including the length of any final update message sent by the manager in Π_{NICP} and any content written to $\mathcal{F}_{\text{VFS}}$, $\mathcal{S}_{\text{NCE}}$ instructs $\mathcal{S}$ to provide the related information and hands it to the environment appropriately.

2. **Simulating the communication phase.** Upon receiving from $\mathcal{F}_{\text{NCE}}$ a message $(\texttt{Sent}, sid, P_S, P_R, |m|)$, $\mathcal{S}_{\text{NCE}}$ sends $(\texttt{Wrote}, sid', o, |m|)$ in the name of $\mathcal{F}_{\text{CRBAC}}$ to $\mathcal{S}$, where $sid' = (M, sid)$. When the environment requests $\mathcal{S}_{\text{NCE}}$ to report the content written to $\mathcal{F}_{\text{VFS}}$, $\mathcal{S}_{\text{NCE}}$ instructs $\mathcal{S}$ to report such content and forwards it as its output appropriately.

3. **Party corruption.** When the environment instructs $\mathcal{S}_{\text{NCE}}$ to corrupt P_S (P_R resp.), $\mathcal{S}_{\text{NCE}}$ delivers the corruption message to $\mathcal{F}_{\text{NCE}}$ and also requests $\mathcal{S}$ to corrupt u_S (u_R resp.). If the corruption happens after P_S has ever sent some message to P_R, $\mathcal{S}_{\text{NCE}}$ will also obtain the messages sent so far from $\mathcal{F}_{\text{NCE}}$. Then it provides the obtained information to $\mathcal{S}$ in the name of $\mathcal{F}_{\text{CRBAC}}$. Once $\mathcal{S}$ outputs the internal state of the corrupt party, $\mathcal{S}_{\text{NCE}}$ forwards it to the environment. After that, any message provided by the environment to the corrupt party would be modified as the message for u_S (u_R resp.) accordingly and forwarded to $\mathcal{S}$ (e.g. if the environment instructs the corrupt sender to send some message c, $\mathcal{S}_{\text{NCE}}$ then instructs $\mathcal{S}$ to write the message c to the file o on behave of u_S). Any request from the environment to corrupt the manager will be ignored.

We briefly analyse the validity of $\mathcal{S}_{\text{NCE}}$. Suppose there exists an environment $\mathcal{Z}$ which can tell its interactions with parties running Π_{NICP} in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model and a dummy adversary, from the interactions with the ideal process for $\mathcal{F}_{\text{NCE}}$ and $\mathcal{S}_{\text{NCE}}$ with non-negligible probability. We show that an environment $\mathcal{Z}'$ can be constructed to tell whether it is interacting with parties running $\Pi_{\mathcal{CRBAC}}$ in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model and a dummy adversary or the interactions with the ideal process for $\mathcal{F}_{\text{CRBAC}}$ and the simulator $\mathcal{S}$ with non-negligible probability. The main idea is that $\mathcal{Z}'$ runs an internal copy of $\mathcal{Z}$ towards which it simulates the view of the ideal process for $\mathcal{F}_{\text{NCE}}$ and the simulator $\mathcal{S}_{\text{NCE}}$. The simulation depends the information that $\mathcal{Z}'$ can obtain during the protocol execution. From the construction of $\mathcal{S}_{\text{NCE}}$ above, it can be inferred that every instruction for $\mathcal{S}_{\text{NCE}}$ can be broken down to corresponding instructions to $\mathcal{S}$. Also, for the inputs that $\mathcal{Z}$ provides to the dummy parties in the ideal process for $\mathcal{F}_{\text{NCE}}$, $\mathcal{Z}'$ can modify them appropriately and provide to the parties it interacts with. Hence we have the simulation $\mathcal{Z}'$ provides to $\mathcal{Z}$ is perfectly identical to the view which $\mathcal{Z}$ expects to see. Then by assumption, $\mathcal{Z}$ can tell its interactions in the two worlds with non-negligible probability, and so can $\mathcal{Z}'$ in this case. Thus, $\mathcal{S}$ cannot be a valid simulator for $\Pi_{\mathcal{CRBAC}}$ which reaches a contradiction. So if $\mathcal{S}$ is a valid simulator for $\Pi_{\mathcal{CRBAC}}$, $\mathcal{S}_{\text{NCE}}$ is also a valid simulator for Π_{NICP} and therefore Π_{NICP} securely realises $\mathcal{F}_{\text{NCE}}$ in $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model.

Now we argue that in fact such a simulator $\mathcal{S}$ does not exist. In [54], it has been shown that no NICP that securely realises $\mathcal{F}_{\text{NCE}}$ exists in the plain model. However, we cannot apply directly that result to complete our proof, since Π_{NICP} makes use of $\mathcal{F}_{\text{VFS}}$ and $\mathcal{F}_{\text{SMT}}$, albeit in a restricted way. Nonetheless, we show that under these usage

restrictions, we can extend Nielsen's result to our setting.

Since Π_{NICP} securely realises $\mathcal{F}_{\text{NCE}}$ in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model, it allows the sender to send arbitrarily many messages to the receiver non-interactively (e.g. by performing write operations to the file system). Any real-world adversary that attacks the protocol cannot obtain more than the length of the transmitted message. Consider the following environment $\mathcal{Z}$. After the communication is established between the message sender P_S and the receiver P_R, $\mathcal{Z}$ activates P_S with an input $(\texttt{Send}, sid, m)$ and requests the adversary to report the content c that has been written to some file o of $\mathcal{F}_{\text{VFS}}$. Once $\mathcal{Z}$ obtains c, it instructs the adversary to corrupt P_R to obtain its internal state st_R. Then $\mathcal{Z}$ produces the current state of the file system from the update information provided by the adversary as fs and computes $m' \leftarrow \mathsf{Read}(st_R, fs, o)$. By assumption $\mathcal{Z}$ should have $m' = m$ except for negligible probability. Then we consider the ideal-world case, the simulator $\mathcal{S}_{\text{NCE}}$ should be able to come up with c given the length of m by $\mathcal{F}_{\text{NCE}}$, and later it should be able to provide the internal state st_R which is consistent to the transmitted message c when m is available by the time P_R is corrupt. Notice that the ideal functionality $\mathcal{F}_{\text{NCE}}$ guarantees correctness on the transmitted message, which means for every message sent by the sender, the receiver should be able to recover the original message except for negligible probability. Hence for Π_{NICP}, there should not exist any local state of the receiver that allows it to decrypt any written content to the file system into two different messages with non-negligible probability each. Otherwise an environment can distinguish its interactions in the two worlds with non-negligible probability. Thus if we fix a file version c, there exists an injective mapping from the underlying messages to the local state of the receiver, which implies that the number of possible internal states st_R of P_R should be at least the same as the number of the possible messages. Notice that the only way P_R can receive the message from P_S is to execute the Read algorithm to retrieve the current content of o from the file system. The injective mapping will not be affected by executing read operations since (by assumption) Read updates neither the file system nor the local state of P_S. Therefore it is impossible for P_R to use the unchanged local state to receiver arbitrary many messages from P_S. Thus we can conclude that Π_{NICP} does not securely realise $\mathcal{F}_{\text{NCE}}$ in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model, which contradicts the existence of the simulator $\mathcal{S}$. Hence there exists no UC-secure $\mathcal{CRBAC}$ (in the $(\mathcal{F}_{\text{VFS}}, \mathcal{F}_{\text{SMT}})$-hybrid model) with adaptive corruptions.

5.5 Conclusion

In this chapter, we present the first security definition for cRBAC systems in the UC framework. Our approach should work for any other model that benefits from a precise semantics with an induced access control matrix. We study the relation between the our security definition and the two existing game-based definitions with respect to secure access, which are presented in the previous chapter. Moreover, we show an impossibility result that no cRBAC system can be UC-secure with adaptive corruptions.

From the above results, we can observe that there is a gap between the two types of security definitions for cRBAC systems. Recall that in [27], the construction of cRBAC system is proven to be secure with respect to both read and write access. Thus, UC security of cRBAC systems is strictly stronger the existing game-based security definitions. However, the existence of the gap is not solely due to the commitment problem which leads to the impossibility result. In fact, UC security does provide stronger security guarantees with respect to both read and write access. To this end, we propose a refinement of the existing game-based definition of read security and also provide a new security definition of write security (both are presented in Chapter 4).

Chapter 6

Some Lower Bounds for secure cRBAC

The work presented in this chapter is adapted from our on-going work on lower bounds for secure cRBAC systems which is not published yet. Some of the theorems are provided with proof ideas only.

6.1 Introduction

Cryptographic access control has received a lot attention in recent decades. However, designing cryptographic access control systems which are of practical use is still a challenging task in this research direction.

Garrison III et al. studied the practical implications of using cryptography to enforce RBAC policies in their recent work [43]. They considered a system model with necessary use of reference monitors to enforce access control on write access and to maintain the metadata of each file in the file system. For their purpose, they developed two different constructions of cryptographic RBAC systems: one bases on identity-based encryption and identity-based signature schemes, while the other one bases on the traditional public key cryptography with the use of Public key infrastructure (PKI). In order to analyse the costs of their constructions, they carried out the simulation over real-world RBAC datasets to generate traces. Their experimental results show that even with the minimum use of reference monitors, the computational costs of the cryptographic RBAC systems which supports for dynamic policy update can be prohibitively expensive.

Motivated by Garrison III et al.'s work, we turn to study lower bounds for secure cRBAC systems to find out where the inefficiency stems from. We show that the costs

are inherent in secure cRBAC systems and also in those cryptographic access control systems that greatly or solely rely on cryptographic techniques to enforce access control on both read and write accesses. The main idea is, since the manager does not involve in any read and write operation to the file system, the local states of the users and also the file system should reflect the access control policy being enforced. Whenever the policy gets updated, the system might inevitably require re-keying and re-encryption in order to guarantee secure access. Our results can be valuable in the design of such systems for practical purposes.

To summarise, in this chapter we make the following contribution:

- we presented two lower bounds for secure cRBAC systems.

6.2 The Lower Bounds

Before we introduce our results, we introduce technical term which we call *Permission Adjustment* for an RBAC system. Informally, permision adjustment is a sequence of RBAC administrative commands which changes the access rights of some user with respect to a set of permissions. In comparison with any sequence of typical RBAC commands, permission adjustment emphasises the change it will bring to the access matrix of the system. Formally:

Definition 14 (Permission Adjustment)**.** *Let $S_0 = (U, O, P, UA, PA)$ be the state of an RBAC system over a set of roles R. Given a set of user $\tilde{U} \subseteq U$ and a set of permissions $\tilde{P} \subseteq P$, where both $\tilde{U}$ and $\tilde{P}$ are non-empty, a sequence of RBAC administrative commands $\vec{q} = (q_0, ..., q_n)$ is called a* ***permission adjustment*** *for S_0 with respect to $\tilde{U}$ and $\tilde{P}$:*

(1) *if $\forall u \in \tilde{U}, p \in \tilde{P} : \neg\mathsf{HasAccess}(u, p)$ holds for S_0 and after a sequence of transitions $S_0 \xrightarrow{q_0}_{\mathcal{S}} S_1 \xrightarrow{q_1}_{\mathcal{S}}, \ldots, \xrightarrow{q_{n-1}}_{\mathcal{S}} S_n \xrightarrow{q_n}_{\mathcal{S}} S_{n+1}$, $\forall u \in \tilde{U}, p \in \tilde{P} : \mathsf{HasAccess}(u, p)$ holds for S_{n+1} or*

(2) *if $\forall u \in \tilde{U}, p \in \tilde{P} : \mathsf{HasAccess}(u, p)$ holds for S_o and after a sequence of transitions $S_0 \xrightarrow{q_0}_{\mathcal{S}} S_1 \xrightarrow{q_1}_{\mathcal{S}}, \ldots, \xrightarrow{q_{n-1}}_{\mathcal{S}} S_n \xrightarrow{q_n}_{\mathcal{S}} S_{n+1}$, $\forall u \in \tilde{U}, p \in \tilde{P} : \neg\mathsf{HasAccess}(u, p)$ holds for S_{n+1}.*

We denote the set of all possible $\vec{q}$ in case (1) by $\tilde{U}{\uparrow}\tilde{P}(S_0)$ and the set of all possible $\vec{q}$ in case (2) by $\tilde{U}{\downarrow}\tilde{P}(S_0)$.

In addition, we introduce two key properties with respect to efficiency.

Definition 15. *Let $st_G = (st_M, fs, \{st_u\}_{u\in U})$ be the global state of a cRBAC system over a set of roles R at some point during its execution. Given a sequence of RBAC administrative commands $\vec{q} = (q_0, ..., q_n)$ and its any implementation $\vec{Q} = (Q_0, ..., Q_n)$ such that for each $0 \leq i \leq n$: Q_i implements the command q_i. After carrying out $\vec{Q}$:*

(1) *if the state of the file system remains unchanged with overwhelming probability, we say that $\vec{q}$ is **file system preserving** for st_G. It is reflected by the following predicate:*

$$\mathsf{FSP}(\vec{q}, st_G) \Leftrightarrow \Pr[\forall \vec{Q} : st_G \xrightarrow{\vec{Q}} st'_G; fs = fs'] = 1,$$

where $st'_G = (st'_M, fs', \{st'_u\}_{u\in U'})$, $\phi(st'_G) = (U', O', P', UA', PA')$ and ϵ is a negligible function in the security parameter;

(2) *if the local states of a set of users $\mathbb{U}$ remain unchanged with overwhelming probability, we say that $\vec{q}$ is $\mathbb{U}$**-user local state preserving** for st_G. It is reflected by the following predicate:*

$$\mathsf{LSP}(\vec{q}, st_G, \mathbb{U}) \Leftrightarrow \Pr[\forall \vec{Q} : st_G \xrightarrow{\vec{Q}} st'_G; \forall u \in \mathbb{U} : st_u = st'_u] = 1,$$

where $st'_G = (st'_M, fs', \{st'_u\}_{u\in U'})$, $\phi(st'_G) = (U', O', P', UA', PA')$, $\mathbb{U} \subseteq U'$ and ε is a negligible function in the security parameter.

Finally, we introduce the concept of *non-trivial execution* for cRBAC system. A non-trivial execution consists of a sequence of operations such that after executing each of the operations in order, for each file in the system, there should exist at least a user that has the read permission for it and also exist at least a user that has the write permission for it. The non-trivial execution serves as a mild assumption on the execution on the cRBAC systems, for the purpose of studying the lower bound of cRBAC systems which are commonly used in practice.

Definition 16. *Let $S_o = (U_0, O_0, P_0, UA_0, PA_0)$ be the initial state of an RBAC system and let $\vec{q} = (q_0, ..., q_n)$ be a sequence of operations. We say $\vec{q}$ is a non-trivial execution if after the sequence of transitions*

$$S_0 \xrightarrow{q_0}_{\mathcal{S}} S_1 \xrightarrow{q_1}_{\mathcal{S}} \ldots \xrightarrow{q_n}_{\mathcal{S}} S_{n+1} = (U_{n+1}, O_{n+1}, P_{n+1}, UA_{n+1}, PA_{n+1}),$$

for each file $o \in O_{n+1}$, there exist users $u, u' \in U_{n+1}$ such that $\mathsf{HasAccess}(u, (o, \mathtt{read}))$ and $\mathsf{HasAccess}(u', (o, \mathtt{write}))$ hold.

Now we present our first necessary lower bound for cRBAC systems which preserve correctness and read security. The theorem states that in a normal execution of a cRBAC system, any (sequence of) RBAC command for cancelling read permissions from users requires update either all the corresponding files or the local states of all the users who have the write permissions to those files.

Theorem 10. *For any cRBAC system which is* ***correct*** *and* ***secure with respect to read access****, it holds that:*

$$\Pr\left[\begin{array}{r} st_G \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R); st_G \xrightarrow{\vec{Q}} st'_G; \forall \vec{q} \in U_r{\downarrow}P_r(\phi(st'_G)) : \\ \mathsf{FSP}(\vec{q}, st'_G) \wedge \mathsf{LSP}(\vec{q}, st'_G, U_w) \end{array}\right] \leq \varepsilon,$$

where $\vec{Q}$ is any non-trivial execution for the system, $st'_G = (st'_M, fs', \{st'_u\}_{u \in U'})$, $\phi(st'_G) = (U', O', P', UA', PA')$, $U_r \subseteq U'$, $P_r \subseteq \{(o, \mathtt{read}) | o \in O'\}$, $U_w = \{u | \forall (o, \mathtt{read}) \in P_r : \mathsf{HasAccess}(u, (o, \mathtt{write}))\}$ and ε is a negligible function in λ.

Proof. We prove the theorem by showing that if the above condition is not satisfied, the cRBAC system cannot be both correct and secure with respect to read accesses. Assume by contradition that there exists a cRBAC system Π which is correct and read secure, while the above condition holds with probability ε_0, which is greater than any ε.

Consider the following adversary $\mathcal{A}$ for $\mathbf{Exp}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda)$. After the random bit b is selected and Π is initialised by running $st_G \leftarrow\!\!\$\ \mathsf{Init}(\lambda, R)$, $\mathcal{A}$ is provided λ and proceeds as follows:

1. $\mathcal{A}$ comes up with a sequence of oracle queries $(\mathsf{query}_0, ..., \mathsf{query}_n)$ for $\mathcal{O}_r$ such that it is equivalent to some non-trivial execution $\vec{Q}$ for Π. $\mathcal{A}$ also comes up with a permission adjustment $\vec{q}$ which is valid for the global state of Π after making the sequence of oracle queries;

2. For each $0 \leq i \leq n$: $\mathcal{A}$ makes the oracle call according to query_i. After that, $\mathcal{A}$ calls FS to obtain the current state of the file system fs;

3. $\mathcal{A}$ calls CorruptU to corrupt a random user $u_0 \in U_r$ to obtain its local state st_{u_0};

4. For each administrative RBAC command in $\vec{q}$, $\mathcal{A}$ calls oracle that corresponds to the command with its argument in order;

5. $\mathcal{A}$ chooses a random file $o \in O$ such that its read permission $(o, \mathtt{read}) \in P_r$. It then calls Challenge with (u_1, o, m_0, m_1), where $u_1 \in U$ is a random user such

that $\mathsf{HasAccess}(u_1, (o, \mathtt{write}))$ holds, m_0 and m_1 are two random messages of the same length. Then $\mathcal{A}$ will obtain the updated state of the file system fs' from the challenge oracle;

6. $\mathcal{A}$ computes $m^* \leftarrow \mathsf{Read}(st_{u_0}, fs', o)$. It terminates with an output 0 if $m^* = m_0$ and 1 if $m^* = m_1$; otherwise, it outputs a random bit $b' \leftarrow_\$ \{0, 1\}$.

Recall that there is an invariant maintained in $\mathbf{Exp}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda)$ to prevent trivial wins: at any point in the experiment, no corrupt user in the list Cr can be granted the read permission of any file in the list Ch. From the construction of $\mathcal{A}$ above, it is clear that when $\mathcal{A}$ calls CorruptU, $Cr = \{u_0\}$ and $Ch = \emptyset$. After carrying out $\vec{q}$, u_0 is no longer authorised to the read permissions in P_r. Hence when $\mathcal{A}$ calls Challenge to specify any file whose read permission is in P_r as its challenge, the invariant will not be violated. Also notice that after carrying out any non-trivial execution, for every file $o \in O$, there should exist at least a user that has its read permission and also a user that has its write permission. Hence the existance of u_1 is guaranteed. Therefore, it can be concluded that $\mathcal{A}$ will not cause any error returned by the oracles.

We now analyse the success probability of $\mathcal{A}$ under $\mathbf{Exp}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda)$. Let $\mathbf{E_0}$ be the event that after $\mathcal{A}$ mades the queries $(\mathsf{query}_0, ..., \mathsf{query}_n)$, Π reaches some global state st'_G such that $\mathsf{FSP}(\vec{q}, st'_G)$ and $\mathsf{LSP}(\vec{q}, st'_G, U_w)$ hold. Let $\mathbf{E_1}$ be the event that $m^* = m_b$. The advantage that $\mathcal{A}$ can gain in the experiment is:

$$
\begin{aligned}
\mathbf{Adv}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda) =& \big| \Pr[\mathbf{Exp}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda) \to \mathtt{true}] - \frac{1}{2} \big| \\
=& \big| \frac{1}{2} \cdot \Pr[\neg \mathbf{E_1}] + \Pr[\mathbf{E_1}] - \frac{1}{2} \big| \\
=& \big| \frac{1}{2} \cdot (\Pr[\neg \mathbf{E_1} \wedge \mathbf{E_0}] + \Pr[\neg \mathbf{E_1} \wedge \neg \mathbf{E_0}]) \\
& + \Pr[\mathbf{E_1} \wedge \mathbf{E_0}] + \Pr[\mathbf{E_1} \wedge \neg \mathbf{E_0}] - \frac{1}{2} \big| \\
=& \big| \frac{1}{2} \cdot (\Pr[\neg \mathbf{E_1} \mid \mathbf{E_0}] \cdot \Pr[\mathbf{E_0}] + \Pr[\neg \mathbf{E_1} \mid \neg \mathbf{E_0}] \cdot \Pr[\neg \mathbf{E_0}]) \\
& + \Pr[\mathbf{E_1} \mid \mathbf{E_0}] \cdot \Pr[\mathbf{E_0}] + \Pr[\mathbf{E_1} \mid \neg \mathbf{E_0}] \cdot \Pr[\neg \mathbf{E_0}] - \frac{1}{2} \big| \leq \varepsilon_1 \qquad (6.1)
\end{aligned}
$$

where ε_1 is a negligible function of λ.

We also consider the following adversary $\mathcal{A}'$ for $\mathbf{Exp}^{\mathrm{corr}}_{\Pi,\mathcal{A}'}(\lambda)$. Here the sequence of queries $(\mathsf{query}_0, ..., \mathsf{query}_n)$ which is equivalent to the non-trivial execution $\vec{Q}$, the user u_0, u_1, the file o and the contents m_0, m_1 are identical to those in $\mathbf{Exp}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda)$ above. The experiment starts from the system initialisation by running Init with the input of security parameter λ and a set of roles R. Then $\mathcal{A}'$ is provided λ and proceeds as follows:

1. For each $0 \leq i \leq n$: $\mathcal{A}'$ call the corresponding oracle to make the query query_i. Then $\mathcal{A}'$ calls FS to obtain the current state of the file system fs;

2. $\mathcal{A}'$ selects a random bit $b \leftarrow\!\!\$\ \{0,1\}$ and then calls the write oracle WRITE with (u_1, o, m_b) and obtains the updated state of the file system fs';

3. $\mathcal{A}'$ terminates with an output (u_0, o).

The challenger then computes $m^* \leftarrow \mathsf{Read}(st_{u_0}, fs^*, o)$. A' wins the game if $m^* \neq m_b$. Let $\mathbf{E_2}$ be the event that Π reaches the global state st'_G and let $\mathbf{E_3}$ be the event that $m^* = m_b$. The advantage that $\mathcal{A}'$ can gain in the experiment is:

$$\begin{aligned}\mathbf{Adv}^{\mathrm{corr}}_{\Pi,\mathcal{A}'}(\lambda) &= \Pr[\mathbf{Exp}^{\mathrm{corr}}_{\Pi,\mathcal{A}'}(\lambda) \rightarrow \mathtt{true}] \\ &= \Pr[\neg \mathbf{E_3}] \\ &= \Pr[\neg \mathbf{E_3} \wedge \mathbf{E_2}] + \Pr[\neg \mathbf{E_3} \wedge \neg \mathbf{E_2}] = 0 \end{aligned} \tag{6.2}$$

From (6.2), it is clear that $\Pr[\neg \mathbf{E_3} \wedge \mathbf{E_2}] = 0$. Hence we have:

$$\begin{aligned}\Pr[\neg \mathbf{E_3} \mid \mathbf{E_2}] \cdot \Pr[\mathbf{E_2}] &= 0 \\ \Pr[\neg \mathbf{E_3} \mid \mathbf{E_2}] &= 0 \end{aligned} \tag{6.3}$$

and

$$\Pr[\mathbf{E_3} \mid \mathbf{E_2}] = 1 - \Pr[\neg \mathbf{E_3} \mid \mathbf{E_2}] = 1 \tag{6.4}$$

Now we relate the advantages of the adversaries in the above two experiments. After carrying out $\vec{Q}$, the tuples (st_{u_0}, st_{u_1}, fs) in the two experiments are identical when the events $\mathbf{E_0}$ and $\mathbf{E_2}$ occur in their individual experiments since the system will reach the same global state st'_G. Notice that in $\mathbf{Exp}^{\mathrm{read}}_{\Pi,\mathcal{A}}(\lambda)$, when both $\mathsf{FSP}(\vec{q}, st'_G)$ and $\mathsf{LSP}(\vec{q}, st'_G, U_w)$ hold, st_{u_1} and fs likely remain unchanged before and after carrying $\vec{q}$. Therefore, after $\mathcal{A}$ calls CHALLENGE and $\mathcal{A}'$ calls WRITE (for the last time), in both cases the file system will be updated by having $fs' \leftarrow\!\!\$\ \mathsf{Write}(st_{u_1}, fs, o, m_b)$. The distributions of fs' in both experiments are identical and fs' will be read with the same user local state st_{u_0}. Then we have:

$$\Pr[\mathbf{E_1} \mid \mathbf{E_0}] = \Pr[\mathbf{E_3} \mid \mathbf{E_2}] \tag{6.5}$$

and

$$\Pr[\neg\mathbf{E_1} \mid \mathbf{E_0}] = \Pr[\neg\mathbf{E_3} \mid \mathbf{E_2}] \tag{6.6}$$

Combining Equations (6.1), (6.3), (6.4), (6.5) and (6.6), we have:

$$\begin{aligned}
\mathbf{Adv}^{\text{read}}_{\Pi,\mathcal{A}}(\lambda) =& \Big|\frac{1}{2}\cdot[0\cdot\varepsilon_0 + \Pr[\neg\mathbf{E_1} \mid \neg\mathbf{E_0}]\cdot(1-\varepsilon_0)] \\
&+ 1\cdot\varepsilon_0 + \Pr[\mathbf{E_1} \mid \neg\mathbf{E_0}]\cdot(1-\varepsilon_0) - \frac{1}{2}\Big| \\
=& \Big|\frac{1}{2}\cdot(1-\Pr[\mathbf{E_1} \mid \neg\mathbf{E_0}])\cdot(1-\varepsilon_0) \\
&+ \varepsilon_0 + \Pr[\mathbf{E_1} \mid \neg\mathbf{E_0}]\cdot(1-\varepsilon_1) - \frac{1}{2}\Big| \\
=& \varepsilon_0 + \frac{1}{2}\cdot\Pr[\mathbf{E_1} \mid \neg\mathbf{E_0}]\cdot(1-\varepsilon_0) \leq \varepsilon_1
\end{aligned} \tag{6.7}$$

From Equation (6.7), we have:

$$\Pr[\mathbf{E_1} \mid \neg\mathbf{E_0}] \leq 2\cdot\frac{\varepsilon_1-\varepsilon_0}{1-\varepsilon_0}$$

Notice that by assumption ε_0 is greater than any negligible function ε, which means $\Pr[\mathbf{E_1} \mid \neg\mathbf{E_0}]$ is negative. Thus, we can conclude that Π cannot be both correct and secure with respect to read accesses in such case, which reaches a contradition. The theorem is therefore proved. □

The following lower bound is for cRBAC systems which preserve correctness and write security. In a similar form, the theorem states that in a normal execution of a cRBAC system, any (sequence of) command which will lead to the revocation of write permissions from users requires update either all the corresponding files or the local states of all the users who have the read permissions to those files. The theorem and its proof only work for the security definition presented in the paper [27] but not for the one presented in this thesis, because the definition presented in Chapter 4 does not require that there exists any user who can read to the file outputed by the adversary and therefore the equation implied by correctness does not hold here. But Definition 10 is strictly stronger than the previous definition, the lower bound is still valuable.

Theorem 11. *For any cRBAC system which is correct and secure with respect to write*

accesses, let $\vec{Q}$ *be its any non-trivial execution, it holds that:*

$$\Pr\left[\begin{array}{l}\forall \vec{Q} : st_G \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R); st_G \xrightarrow{\vec{Q}} st'_G; \forall\ U_w{\downarrow}P_w : \\ \qquad\qquad \mathsf{FSP}(U_w{\downarrow}P_w, st'_G) \wedge \mathsf{LSP}(U_w{\downarrow}P_w, st'_G, U_r)\end{array}\right] \leq \varepsilon,$$

where $st'_G = (st'_M, fs', \{st'_u\}_{u\in U'})$, $\phi(st'_G) = (U', O', P', UA', PA')$, $U_w \subseteq U'$, $P_w \subseteq \{(o, \texttt{write}) | o \in O'\}$, $U_r = \{u | \forall (o, \texttt{write}) \in P_r : \mathsf{HasAccess}(u, (o, \texttt{read}))\}$ *and* ε *is a negligible function in* λ.

Proof sketch. The proof strategy of this theorem is similar to Theorem 10's. Assume by contradicition that the above condition holds with some non-negligible probability, while the system is both correct and secure with respect to write access. Let $\mathcal{A}$, $\mathcal{A}'$ be two adversarys against write security and correctness of the system respectively.

After the two adversaries drive the execution of the cRBAC systems in their individual games according to the non-trivial execution. By assumption, there would exist a user u_0 who has the write permission to a file o and also a user u_1 has the read permission of o (here the adversaries need to make a random guess). Also, the global state of the system will reach the same global state st'_G such that after carrying out the permission adjustment $U_w{\downarrow}P_w$, both $\mathsf{FSP}(U_w{\downarrow}P_w, st'_G)$ and $\mathsf{LSP}(U_w{\downarrow}P_w, st'_G, U_r)$ hold for non-negligible probability.

Then the adversary $\mathcal{A}$ (against write security) requests to corrupt the user u_0 to obtain its local state st_{u_0}. After that, the two adversaries request to carry out the permission adjustments in their games respectively. Notice that at this point, u_0 does not have write permission of o any more. $\mathcal{A}$ then writes some content to o with the local state st_{u_0}. Then $\mathcal{A}$ outputs (u_1, o) and $\mathcal{A}'$ outputs o.

Since the above two conditions hold, the distributions of (st_{u_0}, st_{u_1}, fs) in the two games are identical. Then by assumption, if the cRBAC system is correct, u_1 should be able to read the content written to o by $\mathcal{A}$ with non-negligible probability overall. It therefore leads to a contradicition to the assumption on write security. Thus, the system cannot be both correct and secure with respective to write access in such case. □

6.3 Conclusion

In this chapter, we presented two lower bounds for secure cRBAC systems. To some extent, they mathematically explain the reason why cRBAC systems that support dynamic policy updates may be prohibitively expensive: permission revocation can be

costly. Therefore, for practical purpose one may choose to sacrifice security (e.g. to use lazy re-encryption or support for batch processing) or functionality (e.g. to jointly use other mechanisms to enforce access control policies) to some extent in order to gain efficiency.

There is another lower bound which is not presented in this thesis. It is related to read security and requires some mild assumption on the non-trivial execution. It states that cancelling read permissions from users must result in updating the local states of all the users who have those read permissions. This explains Garrison III's experimental results better but is left as one possible direction for the future work.

Chapter 7

Conclusion

Cryptographic access control, which aims to enforce access control policies with the use of cryptographic techniques, is a promising solution to alleviate the limitations of traditional monitor-based access control. With the emerging trend of outsourcing data storage, there has been considerable interest in this area. However, in the literature of cryptographic access control, formal security models for the whole access control systems have been rarely provided. This leaves a disconnection between the specification of the policies being enforced and the implementation of the cryptographic access control systems. More specifically, since the security of the underlying cryptographic primitives may be overestimated and their use within access control systems may be poorly understood, the absence of formal security models leads to a worrying situation that many of the existing works do not offer formal security proofs for the constructions they proposed.

The starting point of the work in this thesis is the recent study on cryptographic Role-Based Access Control by Ferrara et al. Our main contribution is a comprehensive study of rigorous security models for cRBAC systems.

For the purpose of precisely modelling cryptographic policy enforcement, we study security of cRBAC systems in both game-based and simulation-based settings. We start with proposing security notions of different security properties for cRBAC systems in game-based setting. We believed that our security notions are sufficient for appropriately modelling cryptographic policy enforcement. However, in the follow-up study of cRBAC systems in UC framework, we identify a gap between the UC security notion and our existing game-based notions. Our results show that the UC notion is strictly stronger. Since the UC notion requires the execution of a cRBAC system emulating an ideal process which behaves as a server-mediated access control system, an implementation

of cRBAC system which is considered to be secure in this sense guarantees that the access control policy is always correctly enforced - this is exactly the goal that we want our game-based notions to achieve. Therefore, the existance of the gap has brought us to a question: are the existing game-based security notions appropriately modelling the correct enforcement of the RBAC policy? Unfortunately, there is no definitive answer yet. We also show that no cRBAC implementation can achieve UC security with adaptive party corruption. The impossibility result stems from the well-known commitment problem, which also occurs in the context of cryptographic access control.

Nevertheless, the study of relations between the two types of security notions for cRBAC sytems still gives us some inspiration. We identify two types of attacks which are not captured by the existing game-based security notions. Therefore, we refine the existing notions with respect to secure read and write access respectively in order to capture those overlooked attacks. The refinement work on game-based notions can be seen as a step forward towards our goal, but we still want evidence to confirm that the existing game-based notions are sufficient for modelling correct policy enforcement. The future research on bridging the gap between the two types of security notions may give us the answer and also allow us enjoy the benefits from the two definitional approaches. We will discuss this in the next section.

We also bring forth the study of privacy issues in the context of cryptographic access control systems. We point out that cryptographic implementations may unintentionally leak information about the access control policy being enforced in the system. Indeed, users need to get access to the public available metadata used to implement file system and to the encrypted file contents themselves, yet these may reveal the access policy in place. Therefore, privacy protection becomes an important security requirement of cryptographic access control system, since such information can be sensitive in many application scenarios. We identify and formalise different flavours of policy privacy, targeted to different aspects of such systems. Our results are instructive for the work in similar context.

Finally, we provide a construction of cRBAC system based on a novel type of privacy-preserving predicate encryption and a standard digital signature scheme. We show that our construction securely enforces access control on both read and write access to the file system, while preserving policy privacy to a certain degree.

7.1 Future Work

In this section, we will give several research directions for future work.

Lower bounds for secure cRBAC. A direction for future research is to study the efficiency implications of secure cRBAC systems. Garrison III et al.'s has shown that cryptographic enforcement of dynamic role-based access control policies can be costly [43]. Their findings are based on simulations driven by real-world datasets rather than mathematical proofs. Inspired by their results, we are currently working on the lower bounds for secure cRBAC systems with respect to secure policy enforcement and also preservation of policy privacy. This direction is worth pursuing because it will give us some insights of the efficiency aspects of such systems.

Bridging the gap. The UC security notion for cRBAC systems provides stronger security guarantees, but it requires additional assumption on the underlying encryption scheme due to the well-known difficulty of adaptive security in the UC framework, which renders cRBAC implementations impractical. However, the general composability offered by the UC framework is an important property for cryptographic access control systems. Recall that the system model of cRBAC is in fact a general one, the publicly accessible file storage can consist of encrypted files and metadata which are logically organised as a whole file system. Such data can be used independently by arbitrary application while the policy should still be enforced correctly in such case. On the other hand, our refined game-based notions are seemingly sufficient for capturing cryptographic enforcement of RBAC policy, even though their composability property has not been examined. The game-based notions are sometimes preferable to work with due to its simplicity, especially for complex security tasks like cRBAC systems. Therefore, bridging the gap between the two types of security notions for cRBAC systems can be an interesting research direction since it will allow us enjoy the benefits from the two definitional approaches.

Access Control in Blockchain-based file storage. Blockchain-based technology has received a lot of attention since Bitcoin [53] was launched. As the key feature of blockchain-based systems, the cryptographically auditable, append-only, decentralised transaction ledger has unrolled a wide range of interesting applications. Decentralised file storage is one of the applications that benefit from it. Users who need to rent storage service can pay some fees to hire storage space provided by other users to

store their own data. Currently, several blockchain-based file storage services have been already launched on the market [6, 68, 49] based on various proof systems [45, 10, 62, 29]. In comparison with traditional cloud-based storage service, decentralised file storage can reduce the trust in the service provider while preserving strong data security and also user privacy. The enforcement of access control policies in decentralised file storage would allow for many interesting applications (e.g. granting access and transfering copyright/ownership of digital products, etc.), which are difficult to achieve with traditional cloud-based file storage.

Bibliography

[1] Selim G. Akl and Peter D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Trans. Comput. Syst.*, 1(3):239–248, August 1983.

[2] James Alderman, Jason Crampton, and Naomi Farley. A framework for the cryptographic enforcement of information flow policies. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2017, Indianapolis, IN, USA, June 21-23, 2017*, pages 143–154, 2017.

[3] James P Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, 1972.

[4] Mrinmoy Barua, Xiaohui Liang, Rongxing Lu, and Xuemin Shen. ESPAC: enabling security and patient-centric access control for ehealth in cloud computing. *IJSN*, 6(2/3):67–76, 2011.

[5] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, pages 232–249, 1993.

[6] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. *arXiv preprint arXiv:1407.3561*, 2014.

[7] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 321–334, 2007.

[8] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, pages 213–229, 2001.

[9] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 461–478, 2013.

[10] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54. ACM, 2009.

[11] Xavier Boyen and Brent Waters. Anonymous hierarchical identity-based encryption (without random oracles). In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, pages 290–307, 2006.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.

[13] Ran Canetti and Marc Fischlin. Universally composable commitments. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, pages 19–40, 2001.

[14] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[15] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, pages 453–474, 2001.

[16] Arcangelo Castiglione, Alfredo De Santis, Barbara Masucci, Francesco Palmieri, Aniello Castiglione, and Xinyi Huang. Cryptographic hierarchical access control for dynamic structures. *IEEE Trans. Information Forensics and Security*, 11(10):2349–2364, 2016.

[17] Arcangelo Castiglione, Alfredo De Santis, Barbara Masucci, Francesco Palmieri, Aniello Castiglione, Jin Li, and Xinyi Huang. Hierarchical and shared access control. *IEEE Trans. Information Forensics and Security*, 11(4):850–865, 2016.

[18] Ya-Fen Chang. A flexible hierarchical access control mechanism enforcing extension policies. *Security and Communication Networks*, 8(2):189–201, 2015.

[19] Michael Clear, Arthur Hughes, and Hitesh Tewari. Homomorphic encryption with access policies: Characterization and new constructions. In *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, pages 61–87, 2013.

[20] Stefan Contiu, Rafael Pires, Sébastien Vaucher, Marcelo Pasin, Pascal Felber, and Laurent Réveillère. IBBE-SGX: cryptographic group access control using trusted execution environments. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 207–218, 2018.

[21] Jason Crampton. Practical constructions for the efficient cryptographic enforcement of interval-based access control policies. *CoRR*, abs/1005.4993, 2010.

[22] Jason Crampton. *FAST 2010. Revised selected papers.*, chapter Cryptographic Enforcement of Role-Based Access Control, pages 191–205. 2011.

[23] Jason Crampton. Practical and efficient cryptographic enforcement of interval-based access control policies. *ACM Trans. Inf. Syst. Secur.*, 14(1):14:1–14:30, 2011.

[24] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Over-encryption: Management of access control evolution on outsourced data. In *VLDB*, pages 123–134. ACM, 2007.

[25] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[26] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[27] Anna Lisa Ferrara, Georg Fuchsbauer, Bin Liu, and Bogdan Warinschi. Policy privacy in cryptographic access control. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 46–60, 2015.

[28] Anna Lisa Ferrara, Georg Fuchsbauer, and Bogdan Warinschi. Cryptographically enforced RBAC. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 115–129, 2013.

[29] Ben Fisch, Joseph Bonneau, Nicola Greco, and Juan Benet. Scaling proof-of-replication for filecoin mining. Technical report, Technical report, Stanford University, 2018. https://web. stanford. edu , 2018.

[30] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. *TCC 2016-A, Proceedings, Part II*, chapter Functional Encryption Without Obfuscation, pages 480–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[31] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 197–206, 2008.

[32] David K. Gifford. Cryptographic sealing for information secrecy and authentication. *Commununications of the ACM*, 25(4):274–286, 1982.

[33] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[34] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[35] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 89–98. ACM, 2006.

[36] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 89–98, 2006.

[37] E. Gudes. The Design of a Cryptography Based Secure File System. *IEEE Transactions on Software Engineering*, 6(5):411–420, 1980.

[38] Shai Halevi, Paul A. Karger, and Dalit Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. *IACR Cryptology ePrint Archive*, 2005:169, 2005.

[39] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 158–165. ACM, 2003.

[40] Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. *IACR Cryptology ePrint Archive*, 2011:303, 2011.

[41] Jie Huang, Mohamed A. Sharaf, and Chin-Tser Huang. A hierarchical framework for secure and scalable EHR sharing and access control in multi-cloud. In *41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012*, pages 279–287, 2012.

[42] Luan Ibraimi. Cryptographically enforced distributed data access control. *University of Twente*, 2011.

[43] William C. Garrison III, Adam Shull, Adam J. Lee, and Steven Myers. Dynamic and private cryptographic access control for untrusted clouds: Costs and constructions (extended version). *CoRR*, abs/1602.09069, 2016.

[44] Sonia Jahid, Prateek Mittal, and Nikita Borisov. Easier: encryption-based access control in social networks with efficient revocation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 411–415, 2011.

[45] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.

[46] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[47] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, pages 146–162, 2008.

[48] Ralf Küsters and Max Tuengerthal. The IITM model: a simple and expressive model for universal composability. *IACR Cryptology ePrint Archive*, 2013:25, 2013.

[49] Protocol Labs. Filecoin: A decentralized storage network, 2017.

[50] Benoît Libert and Damien Vergnaud. *Topics in Cryptology – CT-RSA 2009*, chapter Adaptive-ID Secure Revocable Identity-Based Encryption, pages 1–15. 2009.

[51] Bin Liu and Bogdan Warinschi. Universally composable cryptographic role-based access control. In *Provable Security - 10th International Conference, ProvSec 2016, Nanjing, China, November 10-11, 2016, Proceedings*, pages 61–80, 2016.

[52] Hemanta K. Maji, Manoj Prabhakaran, and Mike Rosulek. *Topics in Cryptology – CT-RSA 2011*, chapter Attribute-Based Signatures, pages 376–392. 2011.

[53] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[54] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Advances in Cryptology - CRYPTO 2002*, pages 111–126. Springer Berlin Heidelberg, 2002.

[55] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000.*, pages 245–254, 2000.

[56] Uthpala Subodhani Premarathne, Alsharif Abuadbba, Abdulatif Alabdulatif, Ibrahim Khalil, Zahir Tari, Albert Y. Zomaya, and Rajkumar Buyya. Hybrid cryptographic access control for cloud-based EHR systems. *IEEE Cloud Computing*, 3(4):58–64, 2016.

[57] Saiyu Qi, Yichen Li, Yuanqing Zheng, and Yong Qi. Crypt-dac: Cryptographically enforced dynamic access control in the cloud. *IACR Cryptology ePrint Archive*, 2017:90, 2017.

[58] Mariana Raykova, Hang Zhao, and Steven M. Bellovin. Privacy enhanced access control for outsourced data sharing. In *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, pages 223–238, 2012.

[59] Sushmita Ruj, Milos Stojmenovic, and Amiya Nayak. Privacy preserving access control with authentication for securing data in clouds. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 556–563. IEEE, 2012.

[60] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. American national standard for information technology–role based access control. *ANSI INCITS*, 359:1–49, 2004.

[61] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[62] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107. Springer, 2008.

[63] Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, pages 457–473, 2009.

[64] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003.

[65] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, pages 165–180, 2000.

[66] Guojun Wang, Qin Liu, and Jie Wu. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 735–737, 2010.

[67] Stefan G Weber. Designing a hybrid attribute-based encryption scheme supporting dynamic attributes. *IACR Cryptology ePrint Archive*, 2013:219, 2013.

[68] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014.

[69] Huafei Zhu and Feng Bao. Error-free, multi-bit non-committing encryption with constant round complexity. In *Information Security and Cryptology - 6th International Conference, Inscrypt 2010, Shanghai, China, October 20-24, 2010, Revised Selected Papers*, pages 52–61, 2010.

[70] Yan Zhu, Gail-Joon Ahn, Hongxin Hu, and Huaixi Wang. Cryptographic role-based security mechanisms based on role-key hierarchy. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*, pages 314–319, 2010.